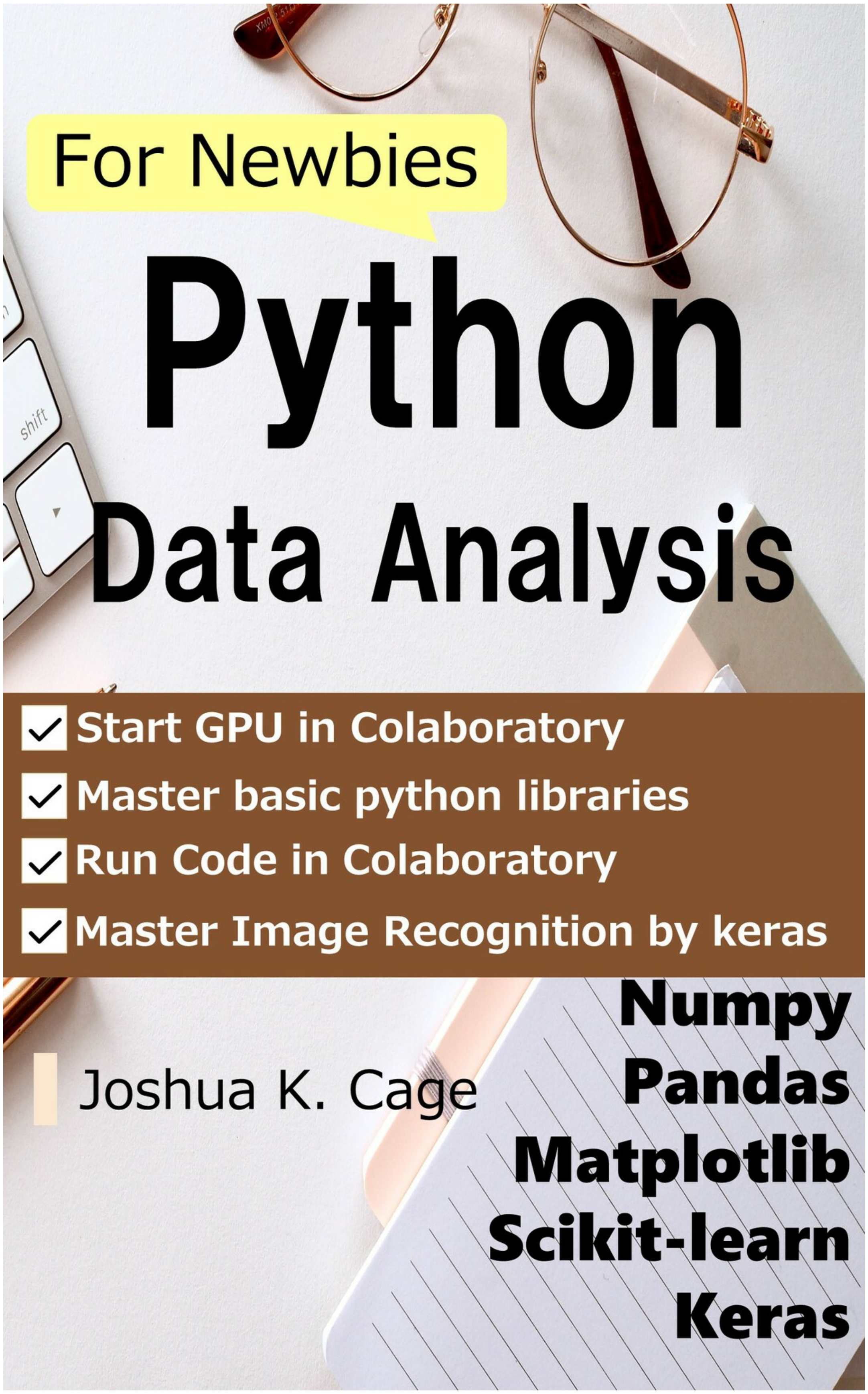
For Newbies

Python Data Analysis

- ✓ Start GPU in Colaboratory
- ✓ Master basic python libraries
- ✓ Run Code in Colaboratory
- ✓ Master Image Recognition by keras

Joshua K. Cage

Numpy
Pandas
Matplotlib
Scikit-learn
Keras



For Newbies

Python Data Analysis

- ✓ Start GPU in Colaboratory
- ✓ Master basic python libraries
- ✓ Run Code in Colaboratory
- ✓ Master Image Recognition by keras

Joshua K. Cage

Numpy
Pandas
Matplotlib
Scikit-learn
Keras

Python Data Analysis for Newbies

Joshua K. Cage

1. Introduction

Thank you for picking up this book. This book is a beginner's introduction to data analysis using Python programming. This book is written for the following readers.

1) Interested in machine learning and deep learning
2) Interested in programming with Python.
3) Interested in data analysis.
4) Interested in using Numpy/Pandas/Matplotlib/ScikitLearn.
5) Not interested in building machine learning environments.
6) Not interested in spending a lot of money for learning.
7) Vaguely worried about the new corona epidemic and the future.

Many of my friends and acquaintances have started data analysis with a vengeance, only to be satisfied with the day-long process of setting up an environment, and then, after doing MNIST (handwritten numeric image data sets) and iris classification tutorials, they get busy with their day jobs and abandon it for a while.

This book uses the free Python execution environment provided by Google to run the tested source code in the book, allowing you to learn by doing programming with zero time to set up your own environment.

This book focuses on the bare minimum of knowledge needed to get a beginner into serious data analysis in Python. Our goal is that by the end of the book, readers will have reached the following five goals.

1) To build and train deep learning models and machine learning models from arbitrary data to be trained and predicted using deep learning libraries (keras) and machine learning libraries (scikit-learn).

2) To use Pandas instead of Excel for large scale data processing.

3) To manipulate multidimensional arrays using Numpy.

4) To draw graphs freely using Matplotlib.

5) To perform simple data analysis on the spread of new coronaviruses.

With the new coronavirus spreading around the world and the various reports in these times of uncertainty about the destination, many of you may not know what to believe and how to go about dealing with the situation.

One thing is for sure, there will be a noticeable difference in the skill sets of individuals depending on how they make use of the new free time created by telecommuting, and we are entering an era of clear winners and losers that will make a huge difference in their value in the company and in the labor market.

I believe that it is vital that we don't continue to hold on to vague fears in a state of anxiety, but rather that we transform each anxiety into a solvable problem through data analysis, one by one, so that each person can choose a course of action.

2. Disclaimer

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 author 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.

You must accept the above precautions before using this book. The author will not be able to respond to inquiries without these precautions. Please be aware that the author will not be able to respond to your inquiry if you do not read these notes.

3. 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.

4. 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

5. 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://drive.google.com/file/d/1G7_YFCGMqV2bfTmR82pSwqLkSxMfhTD
usp=sharing

6. GPU environment ーGoogle Colaboratory

Years ago, programming a Python program for data analysis required setting up a UNIX environment and compiling individual libraries, which was very time-consuming. Nowadays, however, Continuum Analytics offers Anaconda, a Python virtual environment for scientific computing that can be easily installed using an installer and a set of libraries. If you want to set up a Python environment on a local PC, such as a Windows or Mac, you can easily create a stand-alone Python environment using Anaconda.

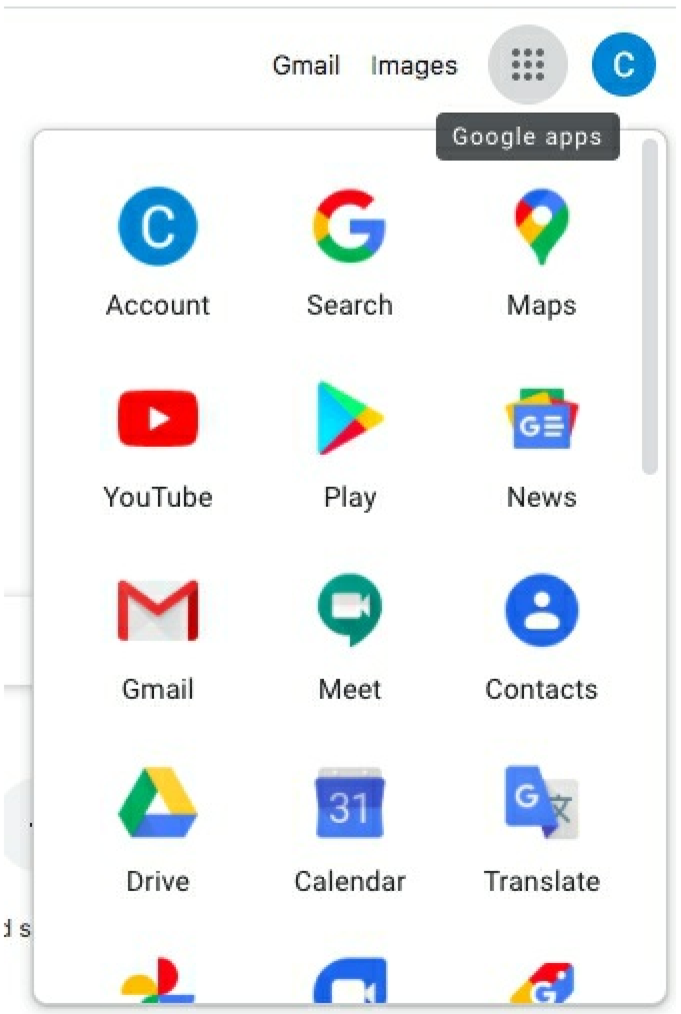
However, Anaconda does have its problems. You need to have your own "local PC". Few novice users have a PC with sufficient specs to run real-world machine learning or deep learning simulations on at home. It wastes a lot of time. This book recommends the use of GPUs in the Google Colaboratory (Colab) environment.

Colab uses a tool called Jupyter Notebook, which is also included with Anaconda, to run Python from a web browser in the cloud (and it's free!). Colab ships standard with Pandas/Numpy/Matplotlib/Keras, which is used in this book. This is a great service that allows you to work on your machine learning projects anytime, anywhere, as long as you have an internet connection, even on a non-powered PC or tablet. With zero risk to get started and zero cost to set up, now that you've picked up a copy of this book you can run your Python programs in Colab and you'll be amazed at how easy it is to write and how quickly you can run deep learning Python programs on GPUs.

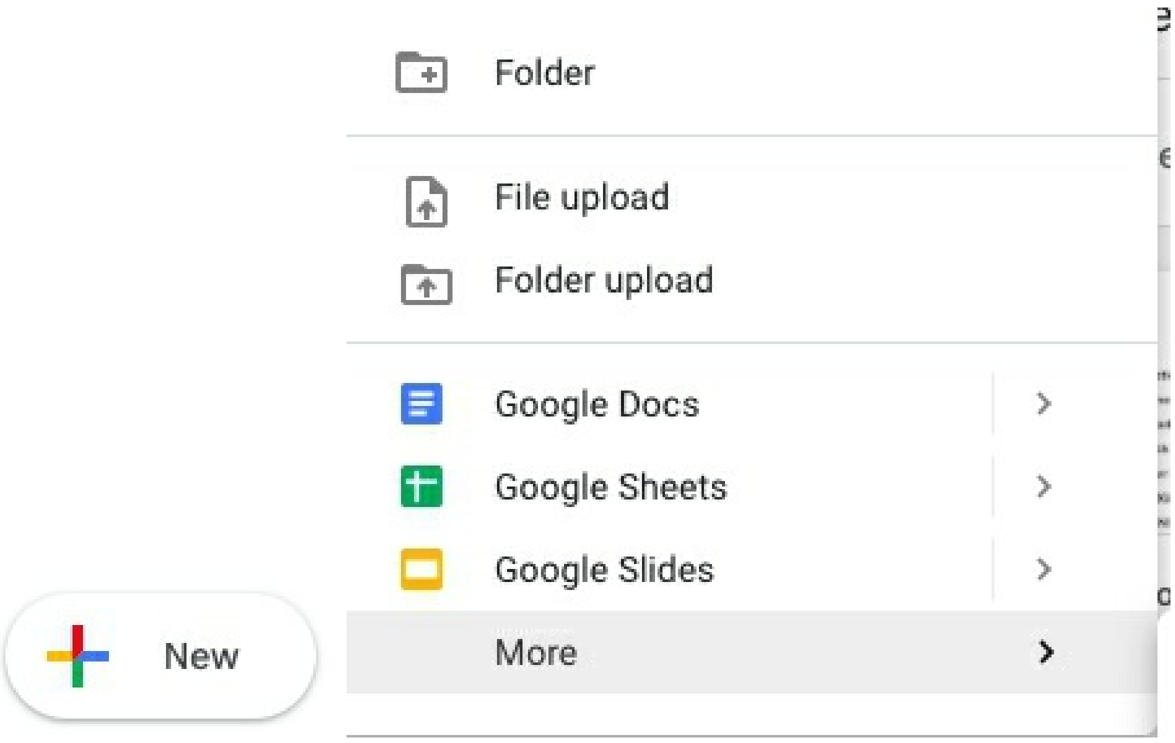
If you don't have a GMAIL account, you will need to create one by clicking on the link here . The following explanation goes on assuming that you already have a gmail account.

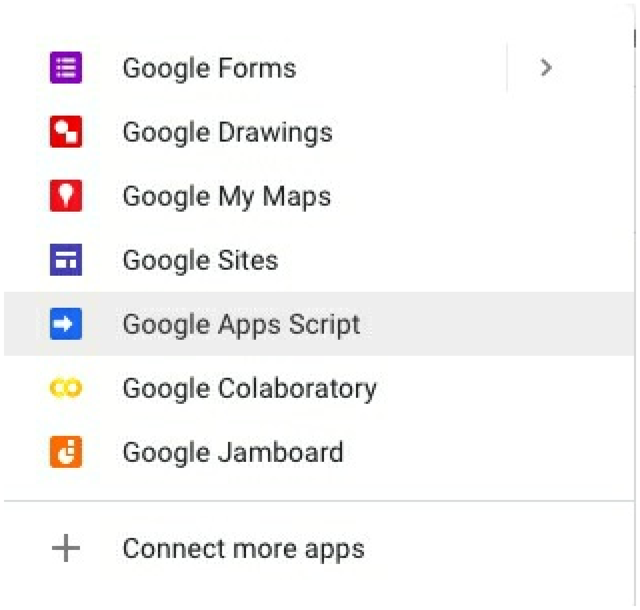
How to Setup Colab

(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.



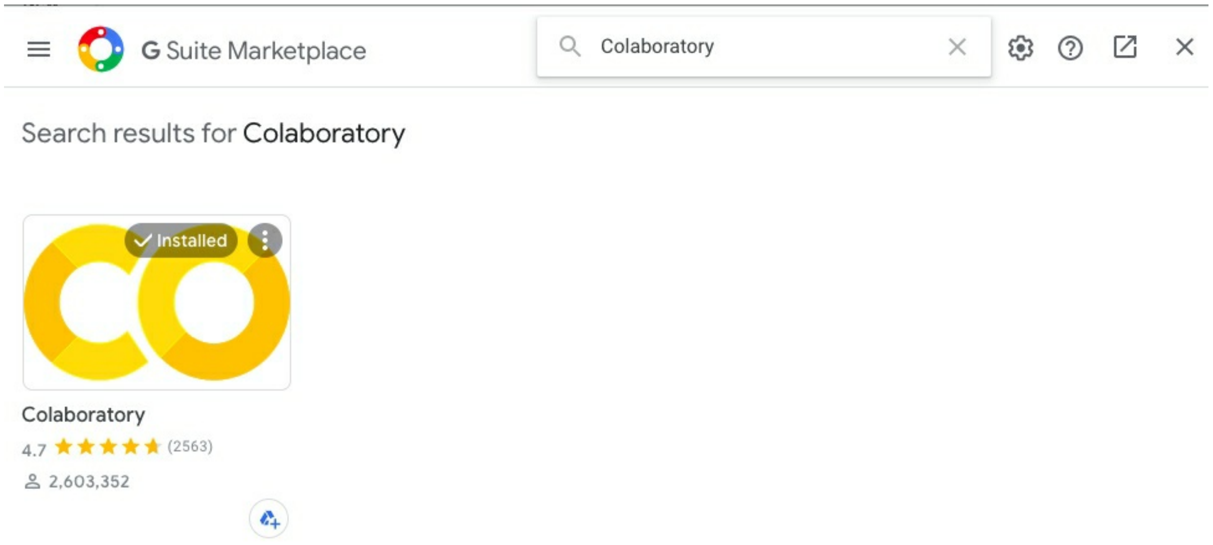
(2) Press the "+ New" button at the bottom of the drive and select "More >" from the menu, then click “Google Colaboratory" if it exists, otherwise choose “Connect more apps".

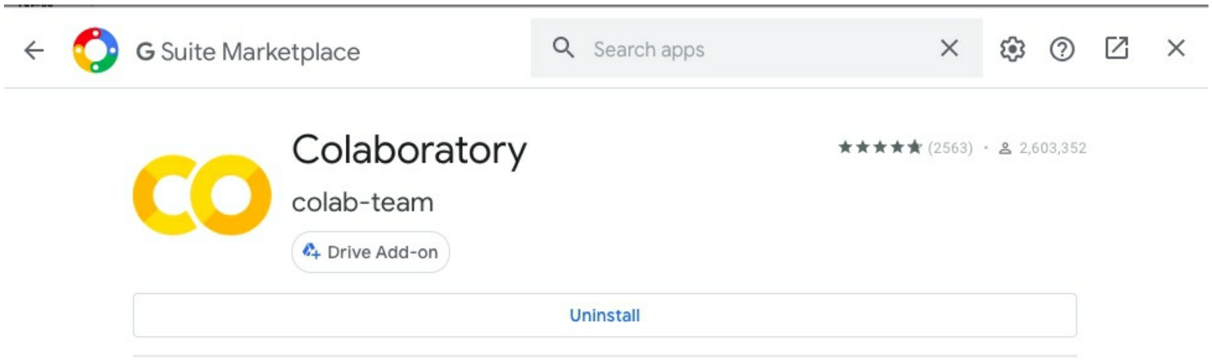




(3) When "G Suite Marketplace" is displayed, click on the magnifying glass mark, and in the text box to search in the app, type "Colaboratory". Please click the "+" button at the bottom right of the logo, and then click the "Install" button on the screen that appears.

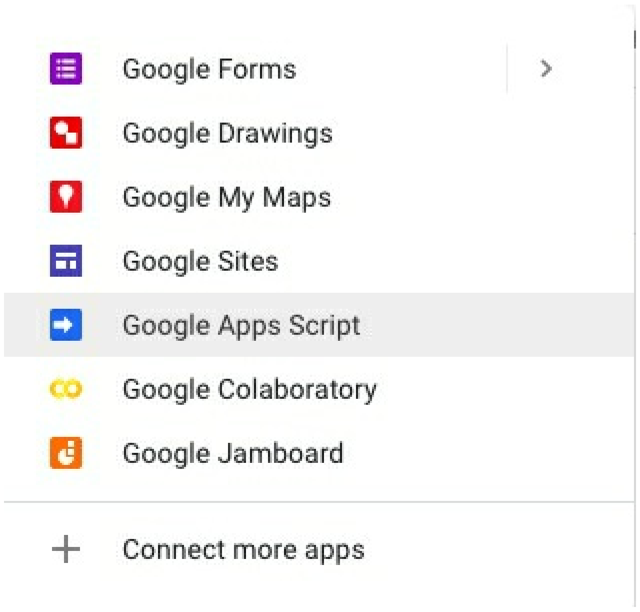
Please click the "Install" button on the screen displayed at the bottom right of the logo.





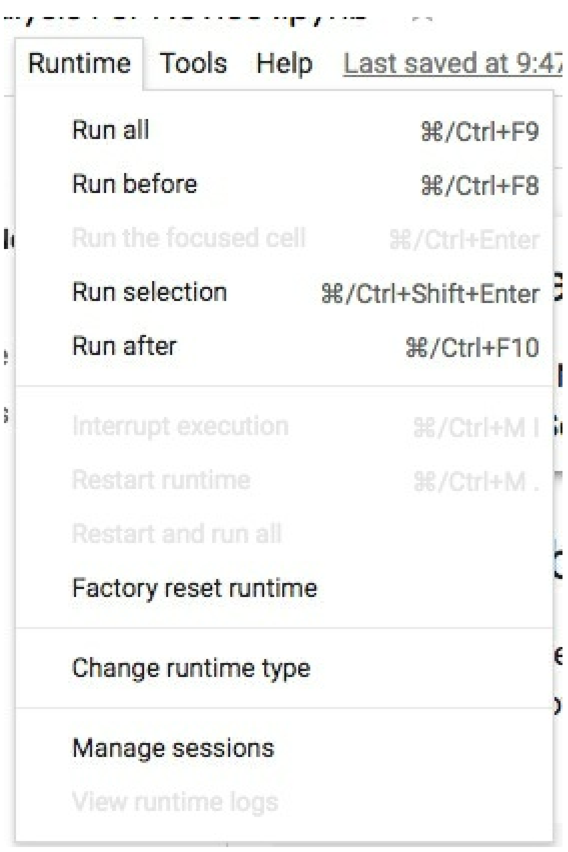
(4) You may be asked to log in again, please continue. When the screen of "Google Colaboratory is now connected to Google Drive. When the screen of "Google Colaboratory has been connected to Google Drive" appears, check the box of "Make Google Colaboratory the default application" and click the "OK" 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.

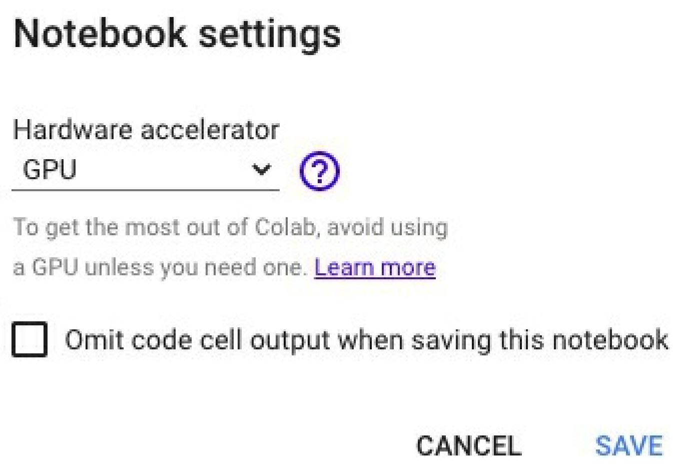
(5) Close the modal window and once again, click the "New +" button and select the "Other >" app. Now you can select "Google Colaboratory".



(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

(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_lib
device_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"]
```

7. Minimal MNIST deeper learning

So let's go ahead and use the GPU to perform deep learning for character recognition of numeric images to get a feel for the speed of the GPU. Enter the following code to train a classifier of 60,000 numbers from 0 to 9, and build a model that looks at the image and predicts what numbers 0-9 are written on it.

i. first, run the code below to import the deep learning library(tensorflow)

```
# Import the required library
from __future__ import absolute_import, division, print_function, unicode_literals

# Import TensorFlow
import tensorflow as tf
```

ii. Load the MNIST numeric image data.

```
# Load MNIST data
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
```

iii. Let's check what kind of data we have by using numpy and matplotlib.

Let's check the MNIST data type.

```
print(type(x_train))
```

Output:

```
<class 'numpy.ndarray'>
```

The MNIST dataset is of the numpy ndarray type, which has the attribute "shape", so by printing the shape, we can see how many records of this data structure are stored.

```
print(x_train.shape)
print(x_test.shape)
print(y_train.shape)
print(y_test.shape)
```

Output:

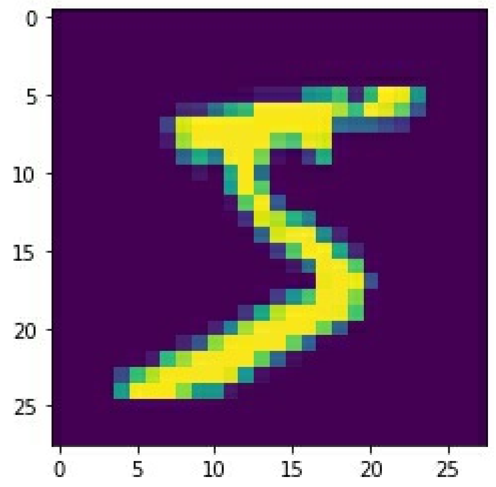
```
(60000, 28, 28)
(10000, 28, 28)
(60000,)
(10000,)
```

As you can see above, x_train.shape and x_test.shape contain 60,000 training datasets of 28 x 28 pixels and 10,000 test datasets of 28 x 28 pixels, respectively. We can also see from y_train.shape and y_test.shape that there are 60,000 correct answer labels for training data and 10,000 for testing, respectively.

You can also use the matplotlib drawing library to draw from a numpy.ndarray (multi-dimensional array object). As a test, let's display a 28 x 28 pixel image of a training dataset.

```
import matplotlib.pyplot as plt
plt.imshow(x_train[ 0 ])
```

Output:



This is an image, but I think you can recognize it as the number 5. I'll also show you the correct answer label that pairs with this image.

```
print(y_train[ 0 ])
```

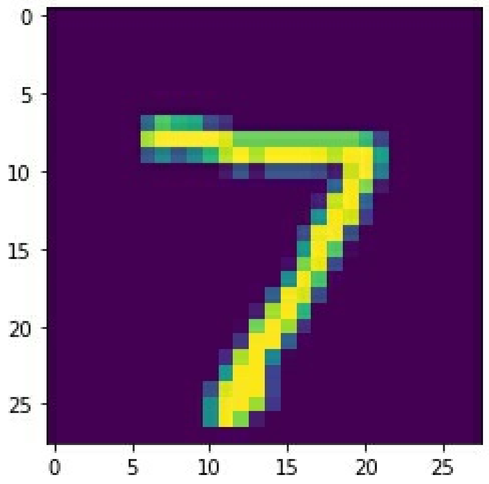
Output:

```
5
```

We could see 5, which we recognized by the image. Then we will view the test data set in the same way.

```
plt.imshow(x_test[ 0 ])
```

Output:



```
y_test[ 0 ]
```

Output:

```
7
```

We will train a deep learning model using the 60,000 training dataset and the pair of correct answer labels x_train and y_train that we just used to see if we can predict the number 7 from the image 7 in this test dataset. We start with the following code to build an all-coupled neural network with three intermediate layers.

```
#  Building a Deep Learning Model
model = tf.keras.models.Sequential([
tf.keras.layers.InputLayer(input_shape=( 784 , )),
tf.keras.layers.Dense( 256 , activation= 'relu' ),
tf.keras.layers.Dropout( 0.2 ),
tf.keras.layers.Dense( 128 , activation= 'relu' ),
tf.keras.layers.Dropout( 0.2 ),
  tf.keras.layers.Dense( 256 , activation= 'relu' ),
tf.keras.layers.Dropout( 0.2 ),
tf.keras.layers.Dense( 10 , activation= 'softmax' )
                                   ])
# Compilation and overview of the model
model.compile(
   optimizer= 'adam' ,
   loss= 'categorical_crossentropy' ,
   metrics=[ 'accuracy' ]
)

print(model.summary())
```

An overview of the built model can be visualized in the following code

```
print(model.summary())
```

Output:

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_14 (Dense)             (None, 256)               200960
_________________________________________________________________
dropout_8 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_15 (Dense)             (None, 128)               32896
_________________________________________________________________
dropout_9 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_16 (Dense)             (None, 256)               33024
_________________________________________________________________
dropout_10 (Dropout)         (None, 256)               0
_________________________________________________________________
dense_17 (Dense)             (None, 10)                2570
=================================================================
Total params: 269,450
Trainable params: 269,450
Non-trainable params: 0
_________________________________________________________________
```

You can see that we have been able to build a model with 269,450 parameters. This is a model of a size that would take a reasonable amount of time to train on the CPU.

We pre-process the data before training it. Since 2D data from (28, 28) cannot be input to the fully coupled neural network as it is, we have to reshape the data into one-dimensional data from (784,). Also, the color information (RGB) is normalized to be between 0 and 1.

```
# Preprocessing of MNIST data (flatten 28 x 28 pixel to 784 for input)
x_train = x_train.reshape( 60000 , 784 )
x_test = x_test.reshape( 10000 , 784 )

# Casting to a float32 type because of a true divide error when dividing by itself
x_train = x_train.astype( 'float32' )
x_test = x_test.astype( 'float32' )

# Normalizes data to a float32 type in the range of 0 to 1
x_train /= 255
x_test /= 255

# Convert to categorical data (1 => [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
y_train = tf.keras.utils.to_categorical(y_train, 10 )
y_test = tf.keras.utils.to_categorical(y_test, 10 )
```

We will train the model with the following code, epochs is the number of times we have to train one training data, but for now we will train it 20 times.

```
model.fit(x_train, y_train, epochs= 20 )
```

Output:

```
Epoch 1/20
1875/1875 [==============================] - 7s 4ms/step - loss:
0.1362 - accuracy: 0.9586
Epoch 2/20
1875/1875 [==============================] - 7s 4ms/step - loss:
0.1105 - accuracy: 0.9673
Epoch 3/20
1875/1875 [==============================] - 7s 4ms/step - loss:
0.0916 - accuracy: 0.9719
Epoch 4/20
1875/1875 [==============================] - 7s 4ms/step - loss:
0.0784 - accuracy: 0.9767
Epoch 5/20
1875/1875 [==============================] - 7s 4ms/step - loss:
0.0702 - accuracy: 0.9788
Epoch 6/20
1875/1875 [==============================] - 7s 4ms/step - loss:
0.0647 - accuracy: 0.9805
Epoch 7/20
1875/1875 [==============================] - 7s 4ms/step - loss:
0.0600 - accuracy: 0.9822
Epoch 8/20
1875/1875 [==============================] - 7s 4ms/step - loss:
0.0579 - accuracy: 0.9830
Epoch 9/20
1875/1875 [==============================] - 7s 4ms/step - loss:
0.0519 - accuracy: 0.9839
Epoch 10/20
1875/1875 [==============================] - 7s 4ms/step - loss:
0.0489 - accuracy: 0.9852
Epoch 11/20
1875/1875 [==============================] - 7s 4ms/step - loss:
0.0477 - accuracy: 0.9855
Epoch 12/20
```

```
1875/1875 [==============================] - 7s 4ms/step - loss:
0.0436 - accuracy: 0.9865
Epoch 13/20
1875/1875 [==============================] - 7s 4ms/step - loss:
0.0429 - accuracy: 0.9870
Epoch 14/20
1875/1875 [==============================] - 7s 4ms/step - loss:
0.0417 - accuracy: 0.9880
Epoch 15/20
1875/1875 [==============================] - 7s 4ms/step - loss:
0.0416 - accuracy: 0.9871
Epoch 16/20
1875/1875 [==============================] - 7s 4ms/step - loss:
0.0377 - accuracy: 0.9887
Epoch 17/20
1875/1875 [==============================] - 7s 4ms/step - loss:
0.0383 - accuracy: 0.9890
Epoch 18/20
1875/1875 [==============================] - 7s 4ms/step - loss:
0.0363 - accuracy: 0.9888
Epoch 19/20
1875/1875 [==============================] - 7s 4ms/step - loss:
0.0346 - accuracy: 0.9900
Epoch 20/20
1875/1875 [==============================] - 7s 4ms/step - loss:
0.0354 - accuracy: 0.9895
```

It may depend on the access time and the speed of the internet, but it takes about 3 minutes to learn. Thanks to Google's GPU, we were able to learn in a super short time, whereas it would have taken us a whole day to learn if we had used a local PC that was not capable of doing so.

The accuracy is a percentage of how many times we got it right, on a scale from 0.0 to 1.0. 1.0 means we got it right 100 times out of 100, while 0.9895 means we got it right about 99 times out of 100 in the training data. You can see that the percentage of correct answers increases with each epoch, from about 95% at the end of the first epoch to nearly 99% at the 20th training session. Incidentally, loss is the error between the output of the neural network and the correct answer given by the training data as an evaluation of the learning process by the neural network. In this chapter, we will use an intuitive and easy to understand percentage of correct answers to explain. The Confusion Matrix and F-values are sometimes used to evaluate the accuracy in situations where severe accuracy measurements are required, such as in research and development. In the chapter on shortest MNIST deep learning, we will only introduce the name of the method.

In the training we just described, you are training on 60,000 training data sets and making predictions in those 60,000 training data sets. This is the first stumbling point for beginners in machine learning and deep learning, but even if you make predictions against the data used for training, you don't know what the percentage of correct answers (i.e., generalization performance) will be against other data you haven't seen yet.

If we look at the following code to evaluate the model against the test data, we see that the percentage of correct answers is 0.9816. We can see that the predictions in the training data were close to 99%, but when we make predictions in the test data, the prediction rate drops by about 1%, and in the MNIST data, the features of the images in the training and test datasets were not that different, so the correctness of the predictions in the training data set and the test. Although the percentage of correct answers in the dataset does not show a large discrepancy, in the world of data science, where learning and prediction are performed on real data, it is not uncommon for training data and test data to have completely different performance results.

```
model.evaluate(x_test, y_test, verbose= 2 )
```

Output:

```
313/313 - 1s - loss: 0.0954 - accuracy: 0.9816
[0.09539060294628143, 0.9815999865531921]
```

So we split the 60,000 data set into 42,000 training data and 18,000 validation data, train it using only 42,000 data, and then see if it can be validated against data that is unknown to the 18,000 models. We call this a "holdout". In implementation, this can be done by simply passing the argument validation_split: 0.3, 30% of the training data set, i.e. 18,000, is used for validation and 80%, i.e. 42,000, is used for training.

```
model.fit(x_train, y_train, epochs= 20 , validation_split= 0.3 )
```

Output:

```
Epoch 1/20
1313/1313 [==============================] - 7s 6ms/step - loss:
0.0249 - accuracy: 0.9931 - val_loss: 0.0107 - val_accuracy: 0.9970
Epoch 2/20
1313/1313 [==============================] - 7s 6ms/step - loss:
0.0256 - accuracy: 0.9937 - val_loss: 0.0102 - val_accuracy: 0.9973
Epoch 3/20
1313/1313 [==============================] - 7s 6ms/step - loss:
0.0251 - accuracy: 0.9935 - val_loss: 0.0083 - val_accuracy: 0.9979
Epoch 4/20
1313/1313 [==============================] - 7s 6ms/step - loss:
0.0274 - accuracy: 0.9929 - val_loss: 0.0145 - val_accuracy: 0.9953
Epoch 5/20
1313/1313 [==============================] - 7s 5ms/step - loss:
0.0227 - accuracy: 0.9933 - val_loss: 0.0099 - val_accuracy: 0.9969
Epoch 6/20
1313/1313 [==============================] - 7s 5ms/step - loss:
0.0243 - accuracy: 0.9938 - val_loss: 0.0170 - val_accuracy: 0.9952
Epoch 7/20
1313/1313 [==============================] - 7s 6ms/step - loss:
0.0244 - accuracy: 0.9939 - val_loss: 0.0166 - val_accuracy: 0.9951
Epoch 8/20
1313/1313 [==============================] - 7s 6ms/step - loss:
0.0230 - accuracy: 0.9936 - val_loss: 0.0173 - val_accuracy: 0.9946
Epoch 9/20
1313/1313 [==============================] - 7s 6ms/step - loss:
```

```
0.0208 - accuracy: 0.9945 - val_loss: 0.0215 - val_accuracy: 0.9938
Epoch 10/20
1313/1313 [==============================] - 8s 6ms/step - loss:
0.0215 - accuracy: 0.9942 - val_loss: 0.0177 - val_accuracy: 0.9953
Epoch 11/20
1313/1313 [==============================] - 8s 6ms/step - loss:
0.0204 - accuracy: 0.9946 - val_loss: 0.0209 - val_accuracy: 0.9945
Epoch 12/20
1313/1313 [==============================] - 8s 6ms/step - loss:
0.0237 - accuracy: 0.9943 - val_loss: 0.0251 - val_accuracy: 0.9932
Epoch 13/20
1313/1313 [==============================] - 7s 6ms/step - loss:
0.0207 - accuracy: 0.9949 - val_loss: 0.0262 - val_accuracy: 0.9935
Epoch 14/20
1313/1313 [==============================] - 7s 6ms/step - loss:
0.0241 - accuracy: 0.9938 - val_loss: 0.0234 - val_accuracy: 0.9937
Epoch 15/20
1313/1313 [==============================] - 7s 6ms/step - loss:
0.0201 - accuracy: 0.9948 - val_loss: 0.0255 - val_accuracy: 0.9937
Epoch 16/20
1313/1313 [==============================] - 7s 6ms/step - loss:
0.0223 - accuracy: 0.9945 - val_loss: 0.0271 - val_accuracy: 0.9929
Epoch 17/20
1313/1313 [==============================] - 7s 5ms/step - loss:
0.0213 - accuracy: 0.9949 - val_loss: 0.0343 - val_accuracy: 0.9919
Epoch 18/20
1313/1313 [==============================] - 7s 6ms/step - loss:
0.0198 - accuracy: 0.9950 - val_loss: 0.0299 - val_accuracy: 0.9927
Epoch 19/20
1313/1313 [==============================] - 7s 6ms/step - loss:
0.0215 - accuracy: 0.9947 - val_loss: 0.0308 - val_accuracy: 0.9929
Epoch 20/20
1313/1313 [==============================] - 7s 6ms/step - loss:
0.0235 - accuracy: 0.9946 - val_loss: 0.0397 - val_accuracy: 0.9906
<tensorflow.python.keras.callbacks.History at 0x7f92e5fa1780>
```

We have added two more metrics to the output, val_loss and val_accuracy, which are the values we evaluated on the 18,000 training data, and we have seen that after 20 training sessions, the val_accuracy is also above 99%.

Now, let's see how well we get the predictions right when we make predictions on data other than the training data set (test data) in a model that has been trained with 42,000 training data.

```
model.evaluate(x_test, y_test, verbose= 2 )
```

Output:

```
313/313 - 1s - loss: 0.1341 - accuracy: 0.9811
```

When we split the 60,000 training data into 42,000 training data and 18,000 verification data, and then train them with the training data and evaluate them against the other 10,000 test data, the percentage of correct answers is 0.9811, that is to say, we predict the handwritten number 100 times and get it correct 98 times You can see that we have built a model that allows us to do this.

We can see that the percentage of correct answers is not that different (in fact, it is slightly lower) than when we had it trained and predicted with the training dataset we just described.

We made the following hypotheses about the cause of this

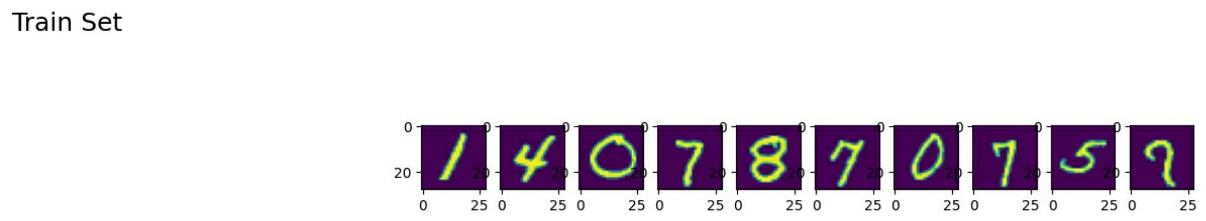
1) Lack of a training dataset.
  → In a training dataset of 42,000 training data, is it not possible to capture the characteristics of the 10,000 training dataset for testing (i.e., there is a lack of variation in the training data)?

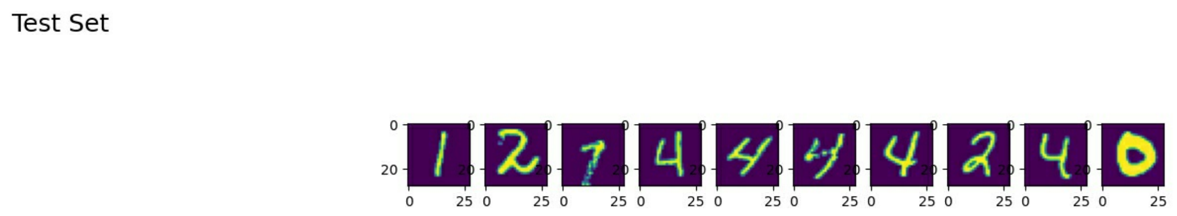
2) Lack of epochs
  →validation_split has reduced the number of data to train the model from 60,000 to 42,000, so the training may still be completed before the peak of the correctness rate?

I'm going to display 10 randomly from each of the training and test data to verify that (1) is true. Earlier, I converted the mnist data to flat in pre-processing, so I'm loading it again.

```
import matplotlib.pyplot as plt
(xo_train, yo_train), (xo_test, yo_test) = tf.keras.datasets.mnist.load_data()

def plot_random10 (mnist_data, title):
""" Get 10 randomly from mnist data and draw
"""
fig = plt.figure(figsize=( 28 , 28 ), dpi= 100 )
fig.suptitle(title, fontsize= 18 , x= 0 , y= 0.92 )
for index, x in enumerate(mnist_data[np.random.choice(np.arange( 0 ,
len(mnist_data)), 10 , replace= False )], 1 ):
   ax = fig.add_subplot( 28 , 28 , index)
   plt.imshow(x)


plot_random10(xo_train, 'Train Set' )
plot_random10(xo_test, 'Test Set' )
```

Output:





By checking, I found that the test data (Test Set) contains more characters that look a little more difficult to decipher than the training data (Train Set). For example, the third data from the right of the Test Set is so blurred that I'm not sure whether it's a 2 or a 3.

Then, to test the hypothesis of 2), I'll use the results of training epochs 100 times to make a prediction. To save space, we will only show the results of the 100th training session.

```
model.fit(x_train, y_train, epochs= 100 , validation_split= 0.3 )
```

Output:

```
1313/1313 [==============================] - 4s 3ms/step - loss:
0.0205 - accuracy: 0.9951 - val_loss: 0.1809 - val_accuracy: 0.9784
```

After the 100th training, the training data set has a correctness rate of 0.9951, while the validation data set has a validation accuracy of 0.9784. It seems that the data set was overfitting to the training data (note: over-fitting to the training data and losing generalization performance). These results suggest that hypothesis (1) seems to make more sense.

Now, we want to solve the next task of predicting using the trained model. With the following code, we can use the test data to perform image recognition.

```
prediction = model.predict(x_test)
```

If we check the predictions in the following code, the result is an array of 10 elements, with real numbers ranging from 0 to 1. The index of the array with the largest number is the predicted number. (The index of the array starts at 0, but you don't need to add 1 because the MNIST numbers are also predicted from 0 to 9.)

```
print(prediction[ 0 ])
```

Output:

```
[1.4635492e-20 4.3042563e-11 7.6560958e-10 1.0590934e-11
5.7983397e-16
5.9631535e-21 2.1780077e-25 1.0000000e+00 3.7560503e-15 3.0590461e-
11]
```

This form is a little confusing, so we can use numpy's argmax function to return the subscript of the largest array of values directly.

```
import numpy as np

print(np.argmax(prediction[ 0 ]))
```

Output:

```
7
```

You can tell if this prediction is correct by outputting the correct answer label.

```
print(np.argmax(y_test[ 0 ]))
```

Output:

```
7
```

So far, we have built a GPU deep learning environment and even used the deep learning library tensorflow to perform image recognition. I think we have probably had the fastest deep learning with GPU environment in a similar book. However, some of you may be able to run the code but have no idea what you're doing, but don't worry, Amazon, which has been a great help to us in Kindle publishing and online shopping in the new Corona era, has a company motto called "Working backwards" and they make the press releases they deliver to their customers first. With that in mind, we used the "get used to it" policy in this book to speed through just a few examples of what you'll be able to do when you learn this book. In the next chapter, I'm going to start explaining the basics of Python and build on the basics to get to the heart of Python data analysis.

8. Python

In the previous chapter, we started with deep learning, which may have left some of you a little bit confused, but don't worry, we're going to start again in this chapter, starting "from scratch". First, let's discuss the basic Python grammar.

Hello Pythonic World!

Python programming is designed for simplicity and is very easy to read and understand, even for beginners. If you are a reader who has worked with other programming languages, you will be surprised at how concise and small the same code is when written in Python.
First, I'd like to touch on this Python philosophy. Run the following code.

```python
import this
```

Output:

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
```

```
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

This is a description of the Python design philosophy.

First of all, try the following code to break your shoulders: "Hello Pythonic World!

```
print( "Hello Pythonic World!" )
```

Output:

```
Hello Pythonic World!
```

Then run the following program: use the Python list (list) to display the English-Japanese translation of the Python philosophy pair. I picked Japanese translation because the first element of the pythonic list is about “Zen” philosophy. “Zen” comes from Japan, “Zen” is basically searching for the truth of who you are. Zen is an abbreviation for Zen Buddhism, a branch of Mahayana Buddhism, or a phonetic transcription of the Sanskrit word ध्यान (dhyāna, dhyana), Zenna.

```
pythonic_english_list = [
"The Zen of Python, by Tim Peters" ,
"Beautiful is better than ugly." ,
"Explicit is better than implicit." ,
"Simple is better than complex." ,
"Complex is better than complicated." ,
"Flat is better than nested." ,
"Sparse is better than dense." ,
"Readability counts." ,
"Special cases aren't special enough to break the rules." ,
"Although practicality beats purity." ,
"Errors should never pass silently." ,
"Unless explicitly silenced." ,
"In the face of ambiguity, refuse the temptation to guess." ,
"There should be one-- and preferably only one --obvious way to do it." ,
"Although that way may not be obvious at first unless you're Dutch." ,
"Now is better than never." ,
"Although never is often better than *right* now." ,
"If the implementation is hard to explain, it's a bad idea." ,
"If the implementation is easy to explain, it may be a good idea." ,
"Namespaces are one honking great idea -- let's do more of those!"
]

pythonic_japanese_list = [
"Python哲学　ティム・ピーターズ" ,
"「綺麗さ」は「汚さ」よりも良い。" ,
"「明示的」は「暗示的」よりも良い。" ,
"「シンプル」は「複雑」よりも良い。" ,
"「込み入っている」よりは「複雑」な方がマシ。" ,
"「ネスト構造」より「フラットな構造」の方が良い。" ,
"「密」より「疎」の方が良い。" ,
"「読みやすさ」が重要。" ,
"特殊なケースはルールを破るほど特別ではない。" ,
"実用性は純粋さに勝る。" ,
"エラーは静かに通過してはいけない。" ,
"明示的に静かにしていない限りは。" ,
"曖昧なコードがあった時、推測して突き進む誘惑に負けないで。" ,
"1つだけ、できれば1つだけ明らかに良い方法があるはずだ。" ,
"その方法はオランダ人でない限り最初はわからないかもしれない。" ,
"今が一番良い。" ,
"しかし「今」は、「たった今」には負ける。" ,
"その実装を説明するのが難しければ、それは悪いアイデアだ。" ,
"その実装を説明するのが簡単であれば、それは良いアイデアかもしれない。" ,
"ネームスペース（名前空間）はとても素晴らしいアイデアなので、もっと使うべきだ。"
]

for english, japanese in zip(pythonic_english_list, pythonic_japanese_list):
```

```
print( f'英語：{english}／日本語：{japanese}' )
```

Output:

```
英語：The Zen of Python, by Tim Peters／日本語：Python哲学　ティム・ピーターズ
英語：Beautiful is better than ugly.／日本語：「綺麗さ」は「汚さ」よりも良い。
英語：Explicit is better than implicit.／日本語：「明示的」は「暗示的」よりも良い。
英語：Simple is better than complex.／日本語：「シンプル」は「複雑」よりも良い。
英語：Complex is better than complicated.／日本語：「込み入っている」よりは「複雑」な方がマシ。
英語：Flat is better than nested.／日本語：「ネスト構造」より「フラットな構造」の方が良い。
英語：Sparse is better than dense.／日本語：「密」より「疎」の方が良い。
英語：Readability counts.／日本語：「読みやすさ」が重要。
英語：Special cases aren't special enough to break the rules.／日本語：特殊なケースはルールを破るほど特別ではない。
英語：Although practicality beats purity.／日本語：実用性は純粋さに勝る。
英語：Errors should never pass silently.／日本語：エラーは静かに通過してはいけない。
英語：Unless explicitly silenced.／日本語：明示的に静かにしていない限りは。
英語：In the face of ambiguity, refuse the temptation to guess.／日本語：曖昧なコードがあった時、推測して突き進む誘惑に負けないで。
英語：There should be one-- and preferably only one --obvious way to do it.／日本語：１つだけ、できれば１つだけ明らかに良い方法があるはずだ。
英語：Although that way may not be obvious at first unless you're Dutch.／日本語：その方法はオランダ人でない限り最初はわからないかもしれない。
英語：Now is better than never.／日本語：今が一番良い。
英語：Although never is often better than *right* now.／日本語：しかし「今」は、「たった今」には負ける。
英語：If the implementation is hard to explain, it's a bad idea.／日本語：その実装を説明するのが難しければ、それは悪いアイデアだ。
```

```
英語：If the implementation is easy to explain, it may be a good idea.／日本語：その実装を説明するのが簡単であれば、それは良いアイデアかもしれない。
英語：Namespaces are one honking great idea -- let's do more of those!／日本語：ネームスペース（名前空間）はとても素晴らしいアイデアなので、もっと使うべきだ。
```

Python list is represented in Python by enclosing them in square brackets[]. It's common for all list elements to be of the same type (all the aforementioned code is of the string type). (All of the above code is of the string type.)

The for statement is a repetition statement. python uses a "for variables in list:" syntax that allows you to store elements in a list of variables and then retrieve them one at a time, starting with the first one. This is called a "for loop". Also, in Python, an indentation is a block of code. The print statement above is indented with two spaces under the for statement, so we can retrieve the elements stored in the variables in the for block.

The syntax for print(f'English:{english}/Japanese:{japanese}') is available in Python version 3.6 and above, and the single quote after the f can be double-quoted. ('' or ""), you can write the name of the variable directly in the braces to interpret and display it as a string. f"{variable }". There are other methods such as the format method of String type and %, but the author recommends using f"{}" notation because it is faster and more visible.

The "List:" part of the for statement can be any iterator type (an object with __iter__ method), so it can be a dictionary type, a tuple type, or even a zip method like the one above that can expand multiple lists in order from the beginning . Let's check if the zip has an __iter__ method, just to be sure.

```
print(zip.__iter__)
```

Output:

```
<slot wrapper '__iter__' of 'zip' objects>
```

Also, I mentioned earlier that it's common to store elements of the same type in all of the lists, but since Python is a language that allocates memory dynamically and is loose on types, it's also possible to put elements of different types - integer, string, floating point, dictionary, tuple, etc. - in one, as shown below.

```
diversity_list = [ 1 , 'One' , 1.0 , { '1' : 'One' }, { 'One' }]
for element in diversity_list:
print(element)
```

Output:

```
1
One
1.0
{'1': 'One'}
{'One'}
```

Most programming languages, except Python, can't achieve such a flexible list, right? Let's display the types in the following code as well.

```
diversity_list = [ 1 , 'One' , 1.0 , { '1' : 'One' }, { 'One' }]
for element in diversity_list:
print(type(element))
```

Output:

```
<class 'int'>
<class 'str'>
<class 'float'>
<class 'dict'>
<class 'set'>
```

Next, let's write and execute a function that returns a string of Python philosophy as you type it in and whether it's in Japanese or English.

```
def get_pythonic_language (texts):
```

```
""" Return which language a pythonic sentence is written in.

   @Args:
     texts(str):input text
    @Returns:
     language(str):language of the input text
"""
pythonic_english_list = [
"The Zen of Python, by Tim Peters" ,
"Beautiful is better than ugly." ,
"Explicit is better than implicit." ,
"Simple is better than complex." ,
"Complex is better than complicated." ,
"Flat is better than nested." ,
"Sparse is better than dense." ,
"Readability counts." ,
"Special cases aren't special enough to break the rules." ,
"Although practicality beats purity." ,
"Errors should never pass silently." ,
"Unless explicitly silenced." ,
"In the face of ambiguity, refuse the temptation to guess." ,
"There should be one-- and preferably only one --obvious way to do it." ,
"Although that way may not be obvious at first unless you're Dutch." ,
"Now is better than never." ,
"Although never is often better than *right* now." ,
"If the implementation is hard to explain, it's a bad idea." ,
"If the implementation is easy to explain, it may be a good idea." ,
"Namespaces are one honking great idea -- let's do more of those!"
]
pythonic_japanese_list = [
"Python哲学　ティム・ピーターズ" ,
"「綺麗さ」は「汚さ」よりも良い。" ,
"「明示的」は「暗示的」よりも良い。" ,
"「シンプル」は「複雑」よりも良い。" ,
"「込み入っている」よりは「複雑」な方がマシ。" ,
"「ネスト構造」より「フラットな構造」の方が良い。" ,
  "「密」より「疎」の方が良い。" ,
  "「読みやすさ」が重要。" ,
  "特殊なケースはルールを破るほど特別ではない。" ,
  "実用性は純粋さに勝る。" ,
  "エラーは静かに通過してはいけない。" ,
  "明示的に静かにしていない限りは。" ,
  "曖昧なコードがあった時、推測して突き進む誘惑に負けないで。" ,
  "１つだけ、できれば１つだけ明らかに良い方法があるはずだ。" ,
  "その方法はオランダ人でない限り最初はわからないかもしれない。" ,
  "今が一番良い。" ,
  "しかし「今」は、「たった今」には負ける。" ,
  "その実装を説明するのが難しければ、それは悪いアイデアだ。" ,
  "その実装を説明するのが簡単であれば、それは良いアイデアかもしれない。" ,
  "ネームスペース（名前空間）はとても素晴らしいアイデアなので、もっと使うべき
だ。"
  ]
  language = ""
  if texts in pythonic_english_list:
    language = "english"
  elif texts in pythonic_japanese_list:
    language = "japanese"
  else :
    language = "not pythonic"
  return language

print(get_pythonic_language( "Now is better than never." ))
```

Output:

```
'english'
```

Here, we use the syntax of a function def statement and an if statement. def statement is also very simple, written as "def function name (argument):" and then in the next indented code block, we write "return return return value" to take some input argument and return the result of the process by If you do not include the return statement, the function will return the special value "None" as a result of executing the function.

The if statement is followed by a conditional expression, and if the conditional expression is met, the code block directly underneath the indented code is executed. If the conditional expression does not match the conditional expression of the if statement but the conditional expression of the elif statement does, the indented code block directly below the else statement is executed. is executed.

"(Double quotation marks) The part enclosed in the three (double quotation marks) should be written for a multi-line comment called a docstring, and the # (hashtag) should be written for a single-line comment. It is ignored at source runtime, but should be filled in to improve the readability of the source.PEP8, the Python coding conventions, also specify how to write a good docstring in PEP257.For the docstring, you can use Google Style and Numpy style is the most common style, but this book will be described using the author's subjective and easy to write Google style.

So far, we've gone over the basic and minimal syntax of defining functions and executing control statements, touching on the Python philosophy. In the next chapter, we'll start to talk about Pandas, a library that can be very useful for analyzing real data as input.

9. Pandas

The significance of Pandas

Pandas is a very powerful tool for data analysis using real data, and although it is similar to EXCEL, spreadsheets and RDB, the benefits of using Pandas are still great!

What makes pandas library stands out from EXCEL is its ability to process large amounts of data at ultra-high speed, especially when it comes to handling 100MB of text data, and it freezes and crashes. It can also run crisply in files, and it supports reading data in EXCEL format.
Python's "for" statements are known to be slow, but Pandas uses Numpy internally, and since Numpy is written in C, it runs as fast as the C language.

DataFrame Type

Before submitting your data to pandas, we first need to prepare the data for dictionary types and list comprehensions. A dictionary type is a data type where Key and Value are paired. In the following example, we define 'Year' and 'Index' Keys and their corresponding list type Values. In the previous chapter, we defined list types one by one using character literals, but this time we use "list comprehension" notation. The syntax of list comprehensions is [Variables for Variables in Iterator Type]. You can now return a list of integers generated by the iterator type, stored in order.

You can use the range function to generate a sequence of integers in a specified range. Passing a single argument yields a list of integers that are incremented by 1 from 0 (the size of the list is generated for the specified number of arguments). If you pass two arguments, you can specify the start and end of the list plus one. For example, you can write [i for i in range(1, 3)] to generate [1, 2]. It's a good idea to remember this as well as to write a list[start:end+1] when you get the specified range of a list.

In 'Year' we could define a list that contains elements that increase by 1 from 2010 to 2030, and in 'Index' we could define a dictionary-type variable year_index that allows lists from 0 to 20.

```
import pandas as pd
year_index = { 'Year' :[y for y in range( 2010 , 2031 )],
              'Index' :[i for i in range( 21 )],
             }
print(year_index)
```

Output:

```
{'Year': [2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019,
2020, 2021, 2022, 2023, 2024, 2025, 2026, 2027, 2028, 2029, 2030],
'Index': [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]}
```

The most common type used in pandas is the DataFrame type, followed by the Series type and, depending on the job, the Panel type, which is not used as much. I'll explain only DataFrame types in this book.

First, let's convert the year_index of the lexicon type we defined earlier to a data frame type for Pandas and store it in a variable.

```
df = pd.DataFrame(year_index)
```

And I'm going to use a handy data frame type method to see what's in the data as easily as it is in EXCEL.

By using DataFrame.head(), we can see the first five records of the data.

```
print(df.head())
```

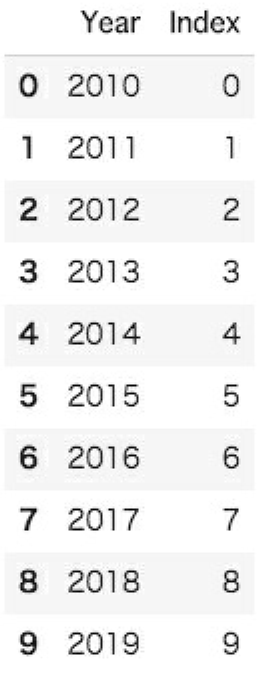
Output:

```
   Year  Index
0  2010      0
1  2011      1
2  2012      2
3  2013      3
4  2014      4
```

I was able to easily display the first five records in table form like this.
As you may have already noticed with the types running in Google Colaboratory and Jupyter Notebook, the print function can display variables without having to write them. The reason we've been writing them explicitly in the past is so that they can be run in a different execution environment (e.g. REPL). If you give an argument to the head, you can display from the beginning to the specified argument.

```
df.head( 10 )
```

Output:

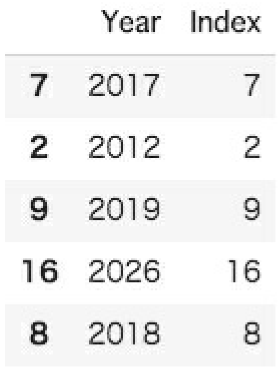


```
   Year  Index
0  2010      0
1  2011      1
2  2012      2
3  2013      3
4  2014      4
5  2015      5
6  2016      6
7  2017      7
8  2018      8
9  2019      9
```

You can use the DataFrame.sample method to randomly fetch and display the records with a specified number of arguments. This is useful when you have data that has a completely different trend from the beginning to the middle or end of the data.

```
df.sample( 5 )
```

Output:



```
    Year  Index
7   2017      7
2   2012      2
9   2019      9
16  2026     16
8   2018      8
```

Next, we'll try out the visualization features of Pandas, but visualizing the Index is no fun, so we'll use a module called random to generate a random number for each element of the Year and store it in a DataFrame type variable, df1.

```
import random
years_random1 = { 'Year' :[y for y in range( 2010 , 2031 )],
                'Random' :[random.random() for y in range( 21 )],
              }
df1 = pd.DataFrame(years_random1)
```

Visualization with just Pandas

Pandas also makes use of matplotlib internally. Therefore, DataFrame type has a method called "plot" which can easily visualize the data stored in it. For example, if you want to display a scatter plot of df1 with Year as the x-axis and Random as the y-axis, the following code will do just fine for simple graphs, such as the xticks to specify the x-axis tick labels.

```
df1.plot(x=df1.columns[ 0 ], y=df1.columns[ 1 ], kind= 'scatter' , xticks=[yea
for year in range( 2010 , 2031 , 5 )])
```

Output:

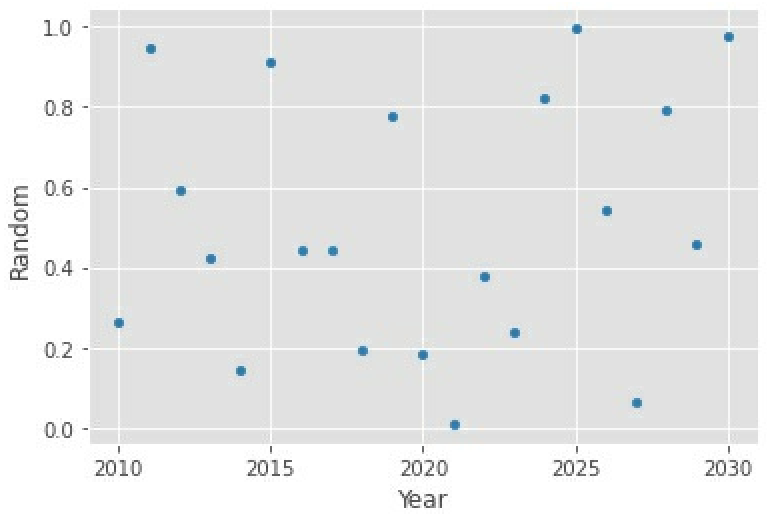


Table joins with a specific key

Next, we'll look at table joins. VLOOKUP and XLOOKUP of EXCEL are also useful, but when the amount of data becomes large, it may freeze instantly. Using Pandas, you can quickly join tables to each other using specific columns as keys.

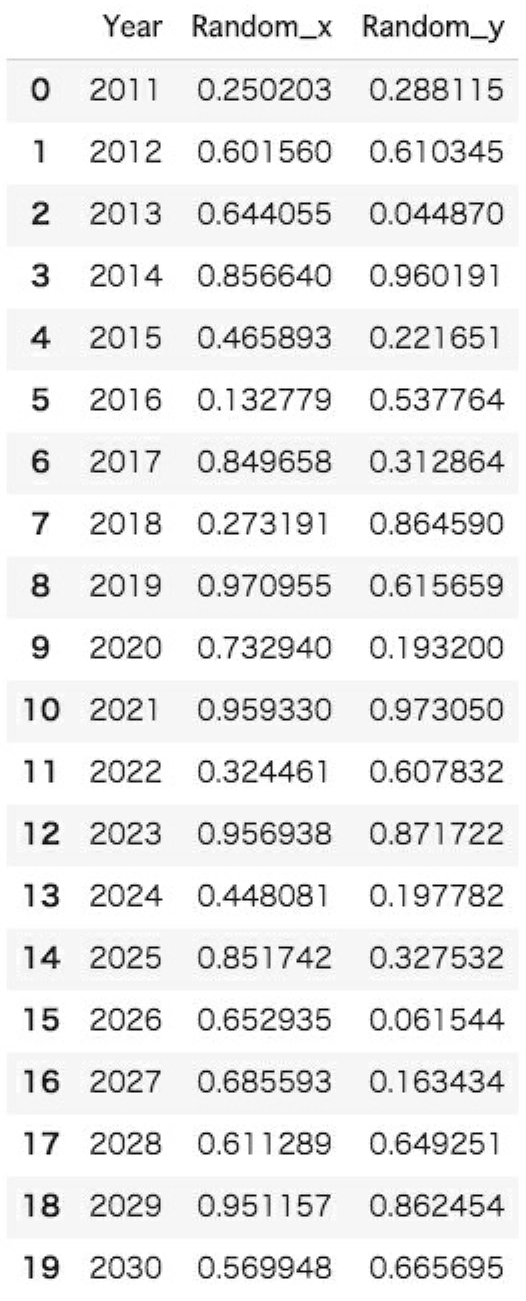
First, we'll have different DataFrames to combine.

```
years_random2 = { 'Year' :[y for y in range( 2011 , 2032 )],
                'Random' :[random.random() for y in range( 21 )],
              }
df2 = pd.DataFrame(years_random2)
```

First, I think the most common way to use this is to use a column as a key and only pull records that exist in both tables, but you can do this as follows. In the following, we are using Year as the key and pulling only records that exist in both tables. Originally, both df1 and df2 had 21 records, but after merging, the number of records is now 20.

```
pd.merge(df1, df2, on= 'Year' , how= 'inner' )
```

Output:

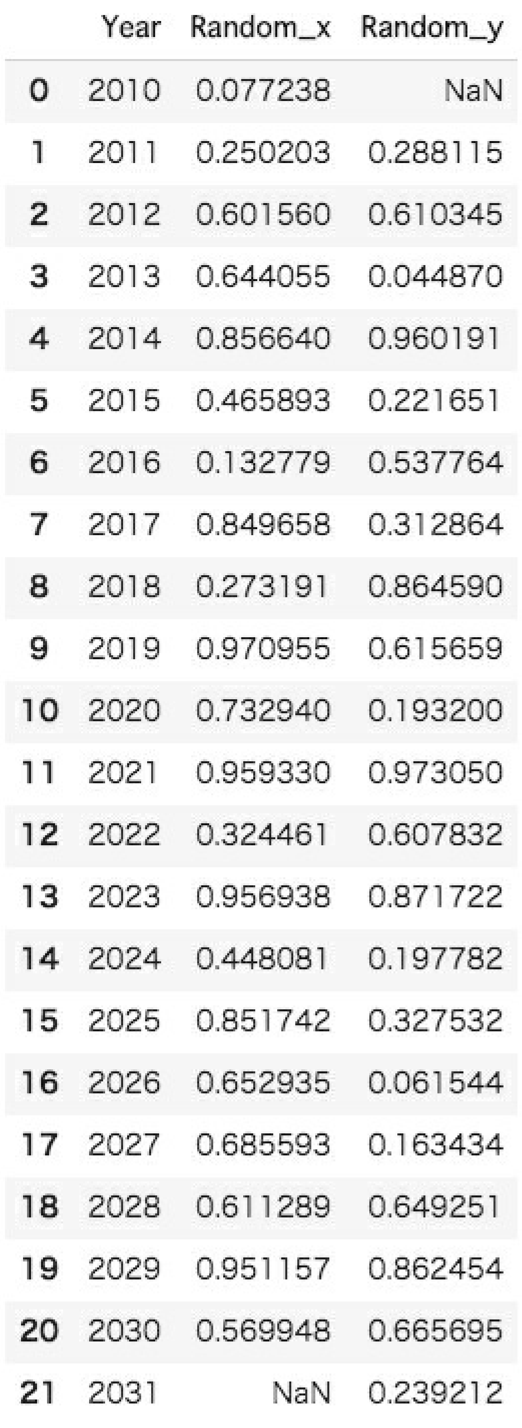


	Year	Random_x	Random_y
0	2011	0.250203	0.288115
1	2012	0.601560	0.610345
2	2013	0.644055	0.044870
3	2014	0.856640	0.960191
4	2015	0.465893	0.221651
5	2016	0.132779	0.537764
6	2017	0.849658	0.312864
7	2018	0.273191	0.864590
8	2019	0.970955	0.615659
9	2020	0.732940	0.193200
10	2021	0.959330	0.973050
11	2022	0.324461	0.607832
12	2023	0.956938	0.871722
13	2024	0.448081	0.197782
14	2025	0.851742	0.327532
15	2026	0.652935	0.061544
16	2027	0.685593	0.163434
17	2028	0.611289	0.649251
18	2029	0.951157	0.862454
19	2030	0.569948	0.665695

An outer join can then be done as follows to extract all the records that exist in one of the two tables, using a column as a key. In the following, we have used Year as a key to pull all the records that exist in either table. Note that the records that exist in only one table are NaNs in the other, and there are 22 records in the table.

```
pd.merge(df1, df2, on= 'Year' , how= 'outer' )
```

Output:

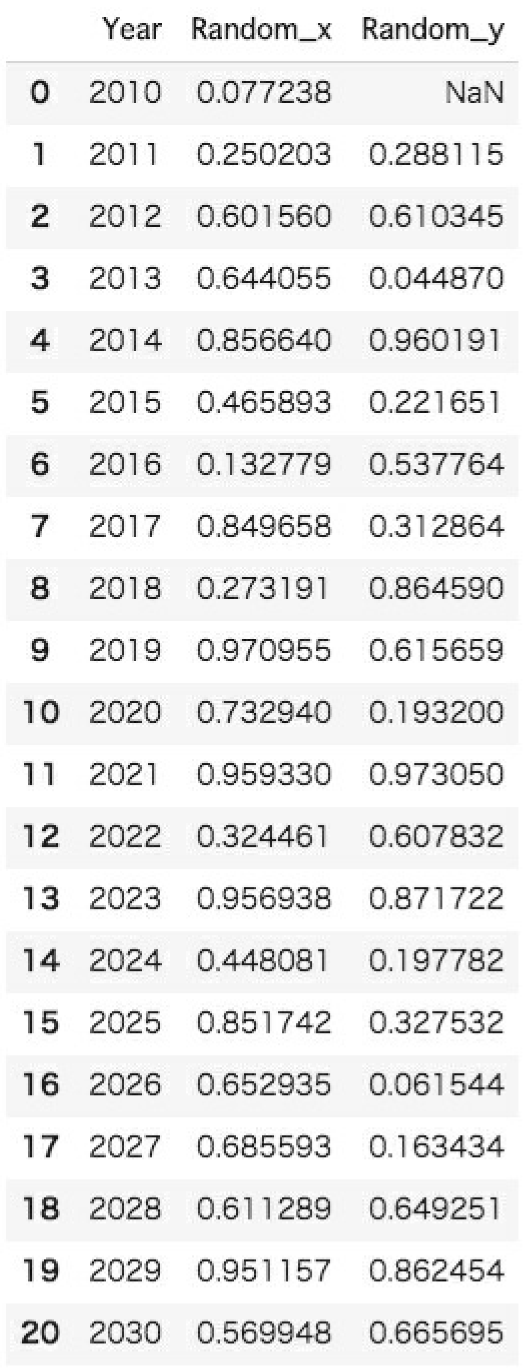


	Year	Random_x	Random_y
0	2010	0.077238	NaN
1	2011	0.250203	0.288115
2	2012	0.601560	0.610345
3	2013	0.644055	0.044870
4	2014	0.856640	0.960191
5	2015	0.465893	0.221651
6	2016	0.132779	0.537764
7	2017	0.849658	0.312864
8	2018	0.273191	0.864590
9	2019	0.970955	0.615659
10	2020	0.732940	0.193200
11	2021	0.959330	0.973050
12	2022	0.324461	0.607832
13	2023	0.956938	0.871722
14	2024	0.448081	0.197782
15	2025	0.851742	0.327532
16	2026	0.652935	0.061544
17	2027	0.685593	0.163434
18	2028	0.611289	0.649251
19	2029	0.951157	0.862454
20	2030	0.569948	0.665695
21	2031	NaN	0.239212

Next, we join df1 and df2 using left outer joins. Note that in the left outer join, we always leave the existing year in df1 intact.

```
pd.merge(df1, df2, on= 'Year' , how= 'left' )
```

Output:

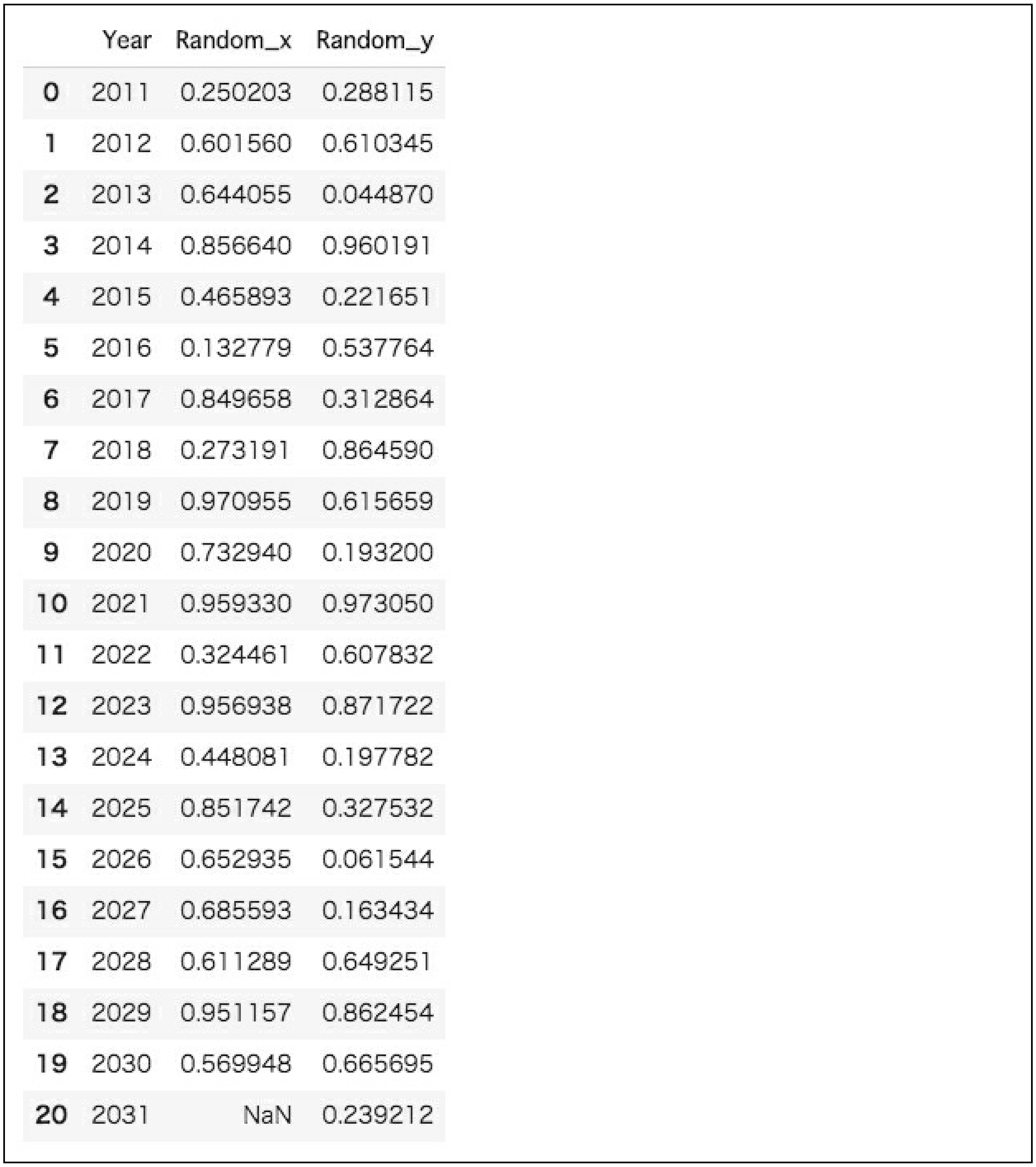


```
    Year  Random_x  Random_y
0   2010  0.077238       NaN
1   2011  0.250203  0.288115
2   2012  0.601560  0.610345
3   2013  0.644055  0.044870
4   2014  0.856640  0.960191
5   2015  0.465893  0.221651
6   2016  0.132779  0.537764
7   2017  0.849658  0.312864
8   2018  0.273191  0.864590
9   2019  0.970955  0.615659
10  2020  0.732940  0.193200
11  2021  0.959330  0.973050
12  2022  0.324461  0.607832
13  2023  0.956938  0.871722
14  2024  0.448081  0.197782
15  2025  0.851742  0.327532
16  2026  0.652935  0.061544
17  2027  0.685593  0.163434
18  2028  0.611289  0.649251
19  2029  0.951157  0.862454
20  2030  0.569948  0.665695
```

Now we'll look at right outer joins, which always leave records that exist in df2.

```
pd.merge(df1, df2, on= 'Year' , how= 'right' )
```

Output:



	Year	Random_x	Random_y
0	2011	0.250203	0.288115
1	2012	0.601560	0.610345
2	2013	0.644055	0.044870
3	2014	0.856640	0.960191
4	2015	0.465893	0.221651
5	2016	0.132779	0.537764
6	2017	0.849658	0.312864
7	2018	0.273191	0.864590
8	2019	0.970955	0.615659
9	2020	0.732940	0.193200
10	2021	0.959330	0.973050
11	2022	0.324461	0.607832
12	2023	0.956938	0.871722
13	2024	0.448081	0.197782
14	2025	0.851742	0.327532
15	2026	0.652935	0.061544
16	2027	0.685593	0.163434
17	2028	0.611289	0.649251
18	2029	0.951157	0.862454
19	2030	0.569948	0.665695
20	2031	NaN	0.239212

CSV file input/output (Colab compatible version)

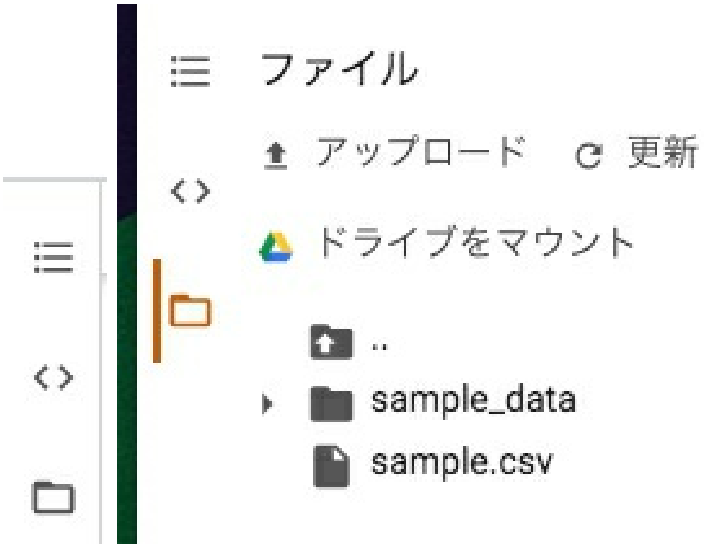
When analyzing real data, you often need to read and write CSV files, and pandas makes it very easy to do so. So much so that people often use Pandas just for the purpose of importing CSV files because it is so intuitive.

Let's take the df1 we've just defined and export it to a CSV file using the method Pandas.to_csv. Google Colab also has a handy feature to mount Google Drive. In this case, let's take a look at a more convenient way of outputting the CSV to an area that is discarded per session.

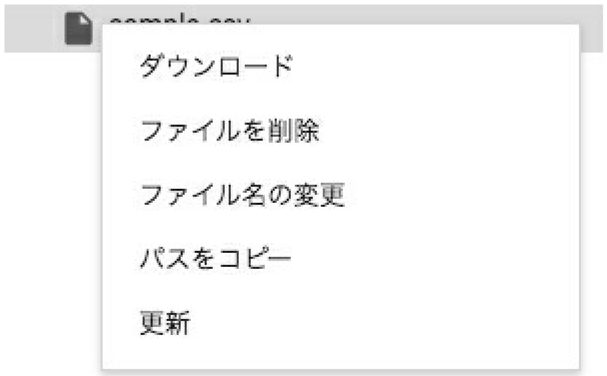
The following code will output a file named sample.csv to the temporary area.

```
df1.to_csv( 'sample.csv' , index= None )
```

Click on the folder icon on the left side of the Colab screen and if the output was successful, you will see a sample.csv.



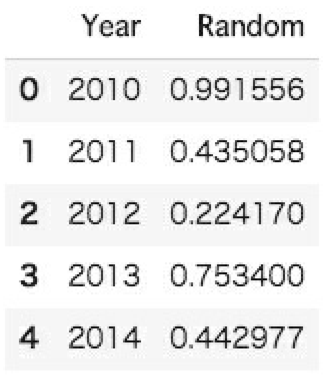
You can also download the output file using the right-click menu.



Next, the CSV file is read with the Pandas.read_csv method as shown below, and by specifying the encoding, we can also read the Windows CP932 encoding file. Since we've just done the to_csv in Colab's Python environment, the output is in UTF-8 and we don't need to specify it, but if we want to read the Windows CSV file, we need to set encoding='cp932' as Please note that df_r.head() also shows the first 5 lines to make sure they are loaded correctly.

```
df1_r = pd.read_csv( 'sample.csv' , encoding= 'utf8' )
df1_r.head()
```

Output:



```
   Year    Random
0  2010  0.991556
1  2011  0.435058
2  2012  0.224170
3  2013  0.753400
4  2014  0.442977
```

Getting any cell or cell range

Let's learn a little more about the basic operations of pandas, just like EXCEL.

If you want to get a specific cell in Pandas, use DataFrame.at or

DataFrame.iat. (The number of rows starts at zero.) The difference between them is that the second argument can be either a column name or an index of the column. The difference between the two is whether the second argument specifies the column name or the column index.

```
dfl_r.at[ 3 , 'Year' ]
```

Output:

```
2013
```

```
dfl_r.iat[ 3 , 0 ]
```

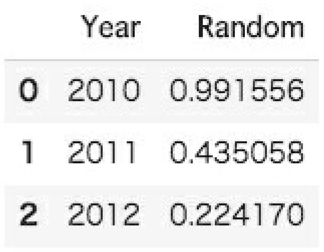
Output:

```
2013
```

Then, if you want to get the values from a range of cells, as in EXCEL, you can use DataFrame.loc or iloc. Like the difference between at and iat, there is a difference between specifying column names and column indices. In addition, you should note that the row index of the loc is different from the slicing of a Python list in that it is a range that includes the endpoint, as you can see in the code below.

```
dfl_r.loc[ 0 : 2 , 'Year' : 'Random' ]
```

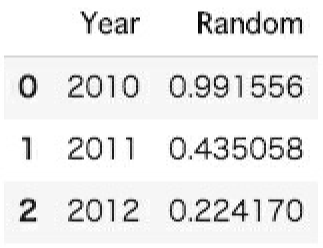
Output:



```
   Year    Random
0  2010  0.991556
1  2011  0.435058
2  2012  0.224170
```

```
dfl_r.iloc[ 0 : 3 , 0 : 2 ]
```

Output:



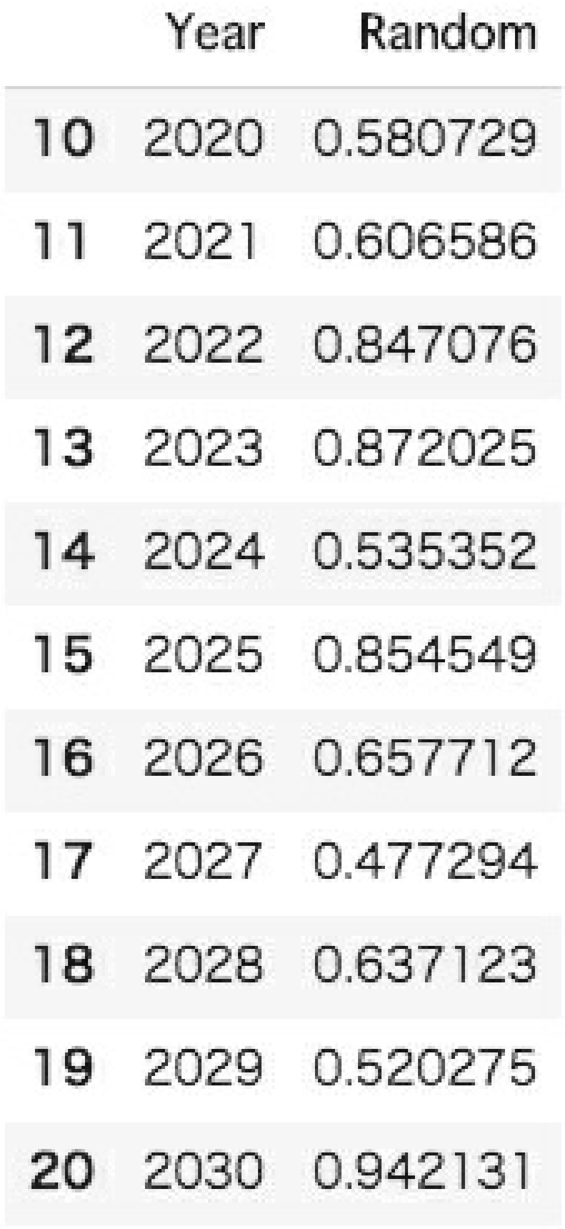
```
   Year    Random
0  2010  0.991556
1  2011  0.435058
2  2012  0.224170
```

Retrieve only the values of the cells that meet specific conditions.

Another convenient way to write DataFrame types is to use the simple df[conditional expression] to retrieve only the records that match the conditions of the pandas DataFrame type. For example, in df1_r, to retrieve only the records from the year 2020 onwards, you can use the following syntax There are two ways to write it, but it means exactly the same thing. Both ways of coding are shown below.

```
df1_r[df1_r.Year>= 2020 ]

df1_r[df1_r[ 'Year' ]>= 2020 ]
```

Output:



	Year	Random
10	2020	0.580729
11	2021	0.606586
12	2022	0.847076
13	2023	0.872025
14	2024	0.535352
15	2025	0.854549
16	2026	0.657712
17	2027	0.477294
18	2028	0.637123
19	2029	0.520275
20	2030	0.942131

Impact of the new corona on stock prices as seen in Pandas

At the end of Pandas, we would like to show a practical example using the pandas-datareader library, which you can install with the following command. By the way, please install Jupyter
In Notebook, if you start with “!” you can directly execute commands in the

environment where Jupyter Notebook is running (Command Prompt for Windows or Terminal for UNIX). It's called "magic command" and it's very useful.

```
!pip install pandas-datareader
```

The pandas-datareader makes it easy to access a variety of sources on the web and retrieve data of the pandas.DataFrame type. The following data sources are supported

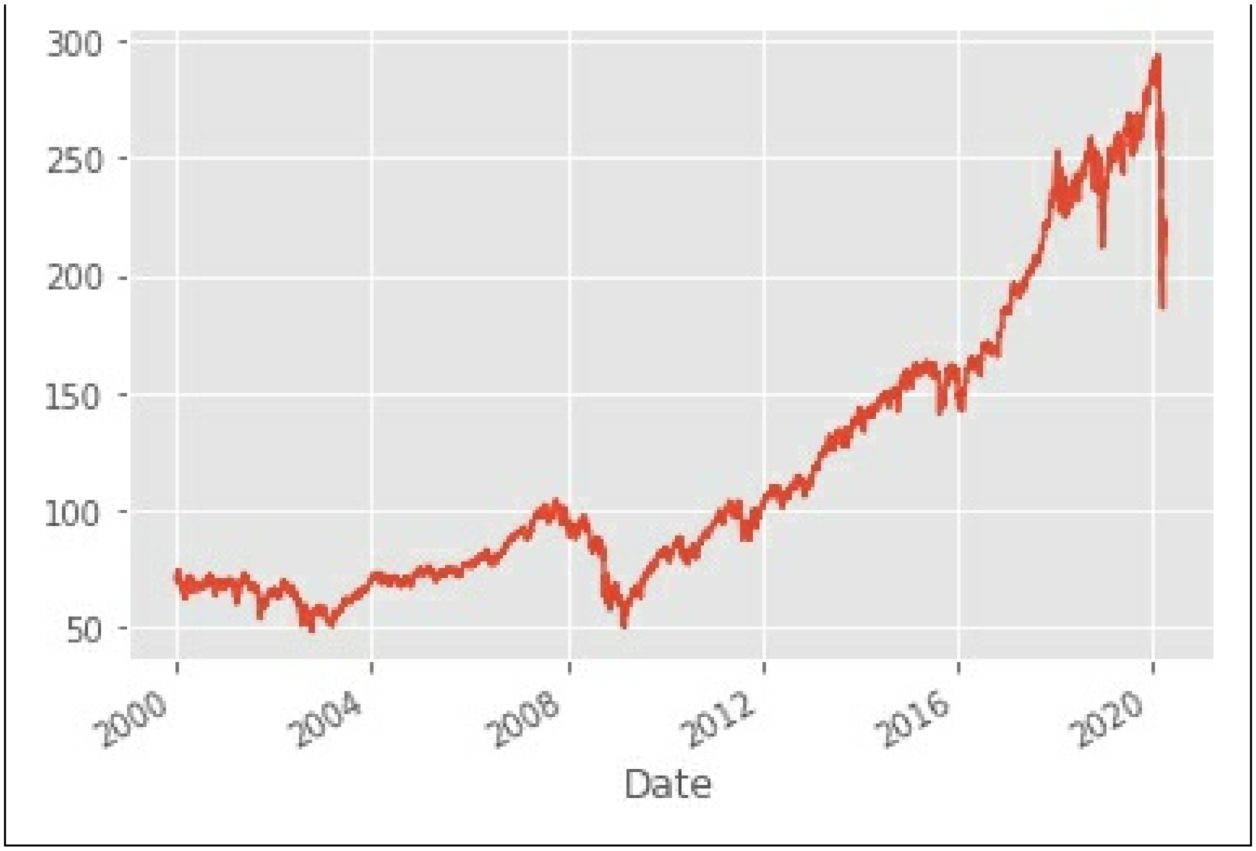
- Tiingo
- IEX
- Alpha Vantage
- Enigma
- Quandl
- St.Louis FED (FRED)
- Kenneth French’s data library
- World Bank
- OECD
- Eurostat
- Thrift Savings Plan
- Nasdaq Trader symbol definitions
- Stooq
- MOEX

The YAHOO Finance API is available as of April 2020, but please refer to the official documentation as it may become unavailable due to API changes. Below is a visualization of the Dow Jones Industrial Average closing price from 1/1/2000 to 4/1/2020. It also shows the last 100 data records.

```
import datetime
import pandas_datareader.data as web
import matplotlib.pyplot as plt
from matplotlib import style

style.use( 'ggplot' )
start = datetime.datetime( 2000 , 1 , 1 )
end = datetime.datetime( 2020 , 4 , 1 )

df = web.DataReader( "dia" , "yahoo" , start, end).dropna()
print(df.tail( 100 ))

df[ 'Adj Close' ].plot()
plt.show()
```

Output:

```
                  High         Low  ...      Volume   Adj Close
Date                                ...
2019-11-07  278.049988  276.559998  ...   3023800.0  273.987885
2019-11-08  277.239990  276.049988  ...   1512000.0  274.007690
2019-11-11  277.410004  275.390015  ...   1729000.0  274.096680
2019-11-12  277.989990  276.640015  ...   1694000.0  274.225220
2019-11-13  278.399994  276.170013  ...   2595600.0  274.976837
...                ...         ...  ...         ...         ...
2020-03-26  225.869995  214.009995  ...  16071900.0  225.070007
2020-03-27  223.259995  214.570007  ...  10545600.0  216.350006
2020-03-30  223.720001  215.100006  ...   7691800.0  223.100006
2020-03-31  224.750000  218.440002  ...   8426600.0  219.229996
2020-04-01  214.779999  207.770004  ...   8570300.0  209.380005

[100 rows x 6 columns]
```



Just by looking here, we can see that the recent impact of the new coronas on the Dow Jones Industrial Average has been remarkable, with a much more violent and direct decline in stock prices than the way they fell during the 2008 Lehman Shock.

10. Numpy

The significance of Numpy

Numpy is a very important library on which other data analysis libraries (e.g., Pandas/Matplotlib/SciPy) are based, and it's one of the main reasons why Python has become such a major language. Numpy allows for super-fast data operations on multidimensional arrays.

Numpy uses a data retention format called ndarray, which, unlike the aforementioned Python lists, can only store data of the same type in ndarray, but instead allocates data continuously in memory (RAM). By storing data in a continuous area of memory, data can be read efficiently in CPU registers.

Another major advantage is that Numpy is linked to linear libraries such as BLAS and LAPACK, which perform vectorization operations on the CPU at compile time, so that you can enjoy the benefits of parallelized high-speed operations without being aware of them.

In order to use numpy, you need to import it with import numpy as np. As of April 2020, Colab seems to have linked to openblas, which we have checked below to see what CPU vectorization libraries are linked and compiled into Numpy.

```
import numpy as np
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)]
```

Now let's take a look at Python list types and how Numpy's ndarray is stored in memory using the id method, which is similar to using the id method to get the address of an allocated memory area. Google Colaboratory's Python is written in the C language (CPython).

Therefore, the value obtained by the id method is an integer value that is cast as an unsigned long pointer to a C programming's PyObject, which is a number with address-like implications.

In the following code, we use the list comprehension notation we learned earlier to generate a list list list1 with 10 random numbers as elements, convert the list to an ndarray type and store it in np_arr.

```
import random
list1 = [random.random() for i in range( 10 )]
np_arr = np.array(list1)
```

The following code then retrieves the index, ID, and element values of np_arr, in the order of the indexes, and displays them

```
for i, elm in enumerate(np_arr):
print(i, id(np_arr[i]), elm)
```

Output:

```
0 140481734505192 0.4730826510610239
1 140481734505144 0.4876796906696862
2 140481734505192 0.9450564589749871
3 140481734505144 0.5117794392437419
4 140481734505192 0.6818941064583345
5 140481734505144 0.5795670075426068
6 140481734505192 0.5898959343675029
```

```
7 140481734505144 0.36767021804934696
8 140481734505192 0.613532458907983
9 140481734505144 0.48151774232816447
```

I see that the memory of np_arr of type ndarray has the same value for 0 and even and the same value for odd numbers. This is because they are stored on a continuous allotment of space.

```
for i, elm in enumerate(list1):
print(i, id(list1[i]), elm)
```

Output:

```
0 140481734504784 0.4730826510610239
1 140481734504904 0.4876796906696862
2 140481734504832 0.9450564589749871
3 140481734505216 0.5117794392437419
4 140481734504808 0.6818941064583345
5 140481734505168 0.5795670075426068
6 140481734505240 0.5898959343675029
7 140481734505264 0.36767021804934696
8 140481734505288 0.613532458907983
9 140481734505312 0.48151774232816447
```

On the other hand, the memory of list type list1 is all different, and we can see that it is stored discontinuously. As python list types are dynamically typed, it's intuitive that storing them like this at distant addresses takes time to retrieve.

Numpy Array Generation Function

Numpy provides a variety of multidimensional array generation features. Keep in mind that this is useful for generating multidimensional arrays of a specific shape with initial values for machine learning and image processing. Numpy.zeros can generate an array with all the elements zero of a given shape. For example, the following will generate an array of 3 rows and 3 columns with all 0's

```
import numpy as np

z = np.zeros(( 3 , 3 ))
print(z)
```

Output:

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

Let's display the summary of the array generated by Numpy.zeros, and you can see that it is a ndarray, 3 x 3 shaped array. Also, each element is an 8 byte itemsize with a float64 type. We can get the number of elements in size and we can get the number of dimensions in ndim. The number of dimensions is the number of axes. A common mistake is to assume that because there are three columns in each row, the number of dimensions is three dimensional. The correct interpretation is that the array is nested (i.e., enclosed in []) and the outermost nest is axis 0, then the outer nest is axis 1, and so on, which is two-dimensional.

```
def print_labeled_val (label, val):
print( f"{label}:{val}" )
def print_ndr_basic_info (z):
print_labeled_val( 'type' , type(z))
print_labeled_val( 'shape' , z.shape)
print_labeled_val( 'itemsize' , z.itemsize)
print_labeled_val( 'size' , z.size)
print_labeled_val( 'dtype' , z.dtype)
print_labeled_val( 'ndim' , z.ndim)

print_ndr_basic_info(z)
```

Output:

```
type:<class 'numpy.ndarray'>
```

```
shape:(3, 3)
itemsize:8
size:9
dtype:float64
ndim:2
```

Since it's important to explain the number of dimensions, we'll use another example to illustrate it a bit further: Numpy.arange allows you to create a one-dimensional array with a start element and an end element of +1, and Numpy.reshape allows you to transform the shape, so the following code defines a 3-dimensional array of 2 x 3 x 4 can be made. In this case, the outermost brackets imply axis 0, so we can see that there are two elements on axis 0. There are also three elements within the second outermost bracket, and the third outermost bracket (i.e., the innermost bracket) has four elements in it. Since there are three pairs of brackets from the outside, it would be easier to remember that ndim=3, representing the 0, 1, and 2 axes from the outside in order.

```
z = np.arange( 0 , 24 ).reshape( 2 , 3 , 4 )
z
```

Output:

```
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
```

The Numpy.zeros I mentioned earlier generated a float64 type by default. If you don't use high-precision scientific calculations, then a float32 type may be sufficient in many cases. The following example shows how to generate Numpy.zeros with a float32 type. The following example shows how to generate Numpy.zeros, which requires half the memory size.

```
z = np.zeros( 10 , dtype= 'float32' )
```

```
print(z.dtype)
print(z.itemsize)
```

Output:

```
float32
4
```

Then generate an array with the elements all initialized at 1 as follows

```
z = np.ones( 10 , dtype= 'float32' )
z
```

Output:

```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=float32)
```

To speed up the generation of arrays, a method to generate multidimensional arrays of the specified form without initializing to 0 or 1 is also shown below.

```
z = np.empty( 10 , dtype= 'float32' )
z
```

Output:

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.], dtype=float32)
```

The following is an example of using Numpy.linspace, which allows you to generate an ndarray from a specified start element to a specified number of end elements, divided evenly into a specified number of elements. In the following example, we have generated an ndarray containing 5 elements, divided by 2 to 10 in increments of 2.

```
z = np.linspace( 2 , 10 , 5 )
z
```

Output:

```
array([ 2.,  4.,  6.,  8., 10.])
```

Tips for Jupyter Notebook

Here's a little trick in Jupyter Notebook, you can add a “?” mark after the object.
When you run the cell, you will see a docstring on how to use it. If you forget how to use it, it is faster than going to the official help page, so you can get through the verification process more efficiently. For example, let's display the help for np.linspace

```
np.linspace?
```

Output:

```
Signature: np.linspace(start, stop, num=50, endpoint=True, retstep=False,
dtype=None, axis=0)
Docstring:
Return evenly spaced numbers over a specified interval.

Returns `num` evenly spaced samples, calculated over the
interval [`start`, `stop`].

The endpoint of the interval can optionally be excluded.

.. versionchanged:: 1.16.0
    Non-scalar `start` and `stop` are now supported.

Parameters
----------
start : array_like
    The starting value of the sequence.
stop : array_like
    The end value of the sequence, unless `endpoint` is set to False.
    In that case, the sequence consists of all but the last of ``num + 1``
    evenly spaced samples, so that `stop` is excluded.  Note that the step
    size changes when `endpoint` is False.
num : int, optional
    Number of samples to generate. Default is 50. Must be non-negative.
endpoint : bool, optional
```

```
    If True, `stop` is the last sample. Otherwise, it is not included.
    Default is True.
retstep : bool, optional
    If True, return (`samples`, `step`), where `step` is the spacing
    between samples.
dtype : dtype, optional
    The type of the output array.  If `dtype` is not given, infer the data
    type from the other input arguments.

    .. versionadded:: 1.9.0

axis : int, optional
    The axis in the result to store the samples.  Relevant only if start
    or stop are array-like.  By default (0), the samples will be along a
    new axis inserted at the beginning. Use -1 to get an axis at the end.

    .. versionadded:: 1.16.0

Returns
-------
samples : ndarray
    There are `num` equally spaced samples in the closed interval
    ``[start, stop]`` or the half-open interval ``[start, stop)``
    (depending on whether `endpoint` is True or False).
step : float, optional
    Only returned if `retstep` is True

    Size of spacing between samples.


See Also
--------
arange : Similar to `linspace`, but uses a step size (instead of the
         number of samples).
geomspace : Similar to `linspace`, but with numbers spaced evenly on a log
            scale (a geometric progression).
logspace : Similar to `geomspace`, but with the end points specified as
           logarithms.
```

```
Examples
--------
>>> np.linspace(2.0, 3.0, num=5)
array([2.  , 2.25, 2.5 , 2.75, 3.  ])
>>> np.linspace(2.0, 3.0, num=5, endpoint=False)
array([2. ,  2.2,  2.4,  2.6,  2.8])
>>> np.linspace(2.0, 3.0, num=5, retstep=True)
(array([2.  ,  2.25,  2.5 ,  2.75,  3.  ]), 0.25)

Graphical illustration:

>>> import matplotlib.pyplot as plt
>>> N = 8
>>> y = np.zeros(N)
>>> x1 = np.linspace(0, 10, N, endpoint=True)
>>> x2 = np.linspace(0, 10, N, endpoint=False)
>>> plt.plot(x1, y, 'o')
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.plot(x2, y + 0.5, 'o')
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.ylim([-0.5, 1])
(-0.5, 1)
>>> plt.show()
File:      /usr/local/lib/python3.6/dist-
packages/numpy/core/function_base.py
Type:      function
```

The ndarray generated by the Numpy.linspace method is also a float64 type, and unlike zeros and ones, you can't specify the initial type of ndarray. So it's important to remember how to change to a float32 type later on.

```
print(z.dtype)
z = z.astype( 'float32' )
print(z.dtype)
```

Output:

```
float64
```

```
float32
```

It is often the case that you want to use random numbers to generate a specified number of integers in a specified range. If you want to reproduce the result (i.e., you want to generate the same value at re-run time = reproducibility), you can fix the seed as follows: The seed argument can be any integer type, but please note that you need to give the same number each time to ensure reproducibility.

In this example we have generated 6 nd arrays for numbers from 0-9.

```
np.random.seed( 0 )
z1 = np.random.randint( 10 , size= 6 )
z1
```

```
array([5, 0, 3, 3, 7, 9])
```

Here's an example of converting from a Python list type to ndarray, which can be done quite easily with the Numpy.array method.

```
b_list = [[ 9 , 8 , 7 ],[ 1 , 2 , 3 ],[ 4 , 5 , 6 ]]
z = np.array(b_list)
z
```

Output:

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

The generated ndarray was found to be a 3 x 3 2D array with each element being of type int64.

```
print_ndr_basic_info(z)
```

Output:

```
type:<class 'numpy.ndarray'>
shape:(3, 3)
itemsize:8
size:9
dtype:int64
ndim:2
```

Numpy.ndarray’s Indexing/Slicing

Retrieving the elements of ndarray is similar to the Python list type, so we'll keep it simple.
First, consider the code to get the first line of code from the z elements of the ndarray type we saw earlier, which were arranged as follows.

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

The ndarray index also starts at 0, so we have the following

```
z[ 0 ]
```

Output:

```
array([9, 8, 7])
```

Next, here's the code to get the elements of the first column of the first row: note that Python's two-dimensional lists use l[0][0], but Numpy.ndarray uses a comma-separated list of rows and columns.

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

Output:

```
9
```

Slicing can be used in the same way as Python lists, except that 2D arrays require a comma-separated row and column range.

```
z[ 0 : 2 , 0 : 2 ]
```

Output:

```
array([[9, 8],
       [1, 2]])
```

To get the last row and column, write the following

```
z[ -1 , -1 ]
```

Output:

```
6
```

Image processing CIFAR10 in Numpy

At the end of the Numpy chapter, let's deal with images in Numpy as a practical example, I'd like to use an RGB color image called CIFAR10.

1.ILSVRC (International Image Recognition Contest) 2012 winner AlexNet's
2. maintained by Alex Krizhevsky
3.RGB color image
4. 10 classes of labels: airplane, automobile, bird, cat, cat, deer, dog, frog, horse, ship, and truck
5.60,000 images (50,000 training images and 10,000 test images)
6. Image size is 32 pixels x 32 pixels
7. Loadable from keras (Tensorflow's wrapper library)

First, let's use keras to load the cifar10 data. The loaded datasets are all of type ndarray, and we can see that there are 50,000 training data and 10,000 test data for 32x32 pixels of RGB data. We can see that the correct label has an int64 type value in it.

```
from keras.datasets import cifar10
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
```

```
print_ndr_basic_info(x_train)
print_ndr_basic_info(x_test)
print_ndr_basic_info(y_train)
print_ndr_basic_info(y_test)
```

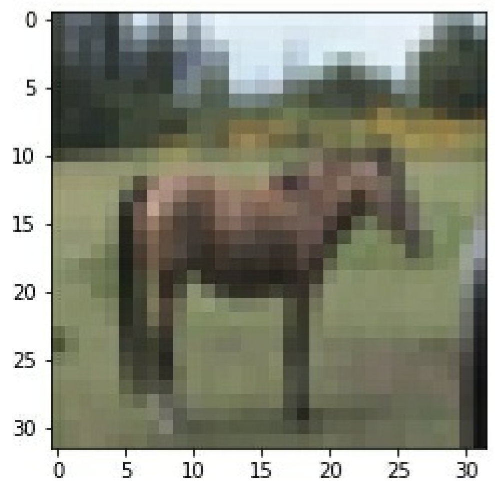
Output:

```
<class 'numpy.ndarray'>
(50000, 32, 32, 3)
1
153600000
uint8
4
<class 'numpy.ndarray'>
(10000, 32, 32, 3)
1
30720000
uint8
4
<class 'numpy.ndarray'>
(50000, 1)
1
50000
uint8
2
<class 'numpy.ndarray'>
(10000, 1)
8
10000
int64
2
```

Since the data is image data, it can be easily displayed with the following code using matplotlib, the library we will discuss in the next section. Try to display the 10,000th record of the image in the test data. Horse" is now displayed.

```
from matplotlib import pyplot as plt
plt.imshow(x_test[ 9999 ])
```

Output:



Slicing in Numpy.ndarray could be done like a list. So outputting the axis 0 (i.e. the rows) in reverse order is done as in z[::-1]. For example, doing this for a 3 x 3 two-dimensional array would result in the following

```
z = np.array([[ 1 , 2 , 3 ],[ 4 , 5 , 6 ],[ 7 , 8 , 9 ]])
z[:: -1 ]
```

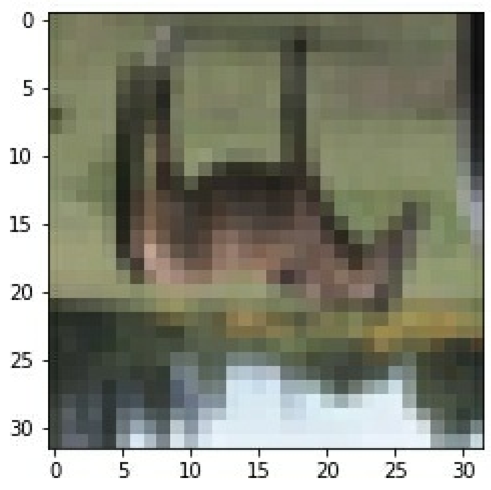
Output:

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

If you view the result of processing this against the image of the horse I just showed you, you will see an upside down horse in the image. It's intuitive and easy to understand.

```
plt.imshow(x_test[ 9999 ][:: -1 ])
```

Output:



Then, outputting axis 1 (i.e., the column) in reverse order is done as follows

```
z = np.array([[ 1 , 2 , 3 ],[ 4 , 5 , 6 ],[ 7 , 8 , 9 ]])
z[:, :: -1 ]
```

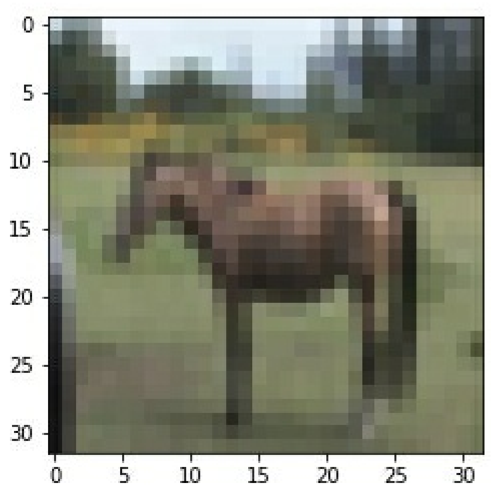
Output:

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

If you process and display this against the image of the horse from earlier, you will see a horse whose front and back are reversed this time. Again, the result is very easy to understand.

```
plt.imshow(x_test[ 9999 ][:, :: -1 ])
```

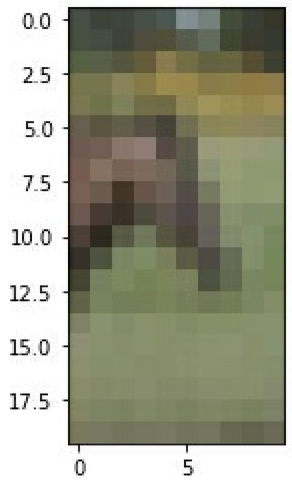
Output:



The code to slice out the head of the horse image is shown below. By slicing both rows and columns (more precisely, both axis 0 and axis 1), we can easily achieve the process of slicing out a portion of the image area.

```
plt.imshow(x_test[ 9999 ][ 5 : 25 , 20 : 30 ])
```

Output:



Next, I'd like to look at compression: just like Slicing for Python list types, Numpy allows you to generate an array by specifying an interval (how many every other one). For example, the following example creates a new array by skipping the rows and columns of the original array one by one. The shape is reduced from 5 x 5 to 3 x 3.

```
z = np.array([[ 1 , 2 , 3 , 4 , 5 ],[ 6 , 7 , 8 , 9 , 10 ],[ 11 , 12 , 13 , 14 , 15 ],[ 16
22 , 23 , 24 , 25 ]])
print(z)
print(z.shape)
print(z[:: 2 , :: 2 ])
print(z[:: 2 , :: 2 ].shape)
```

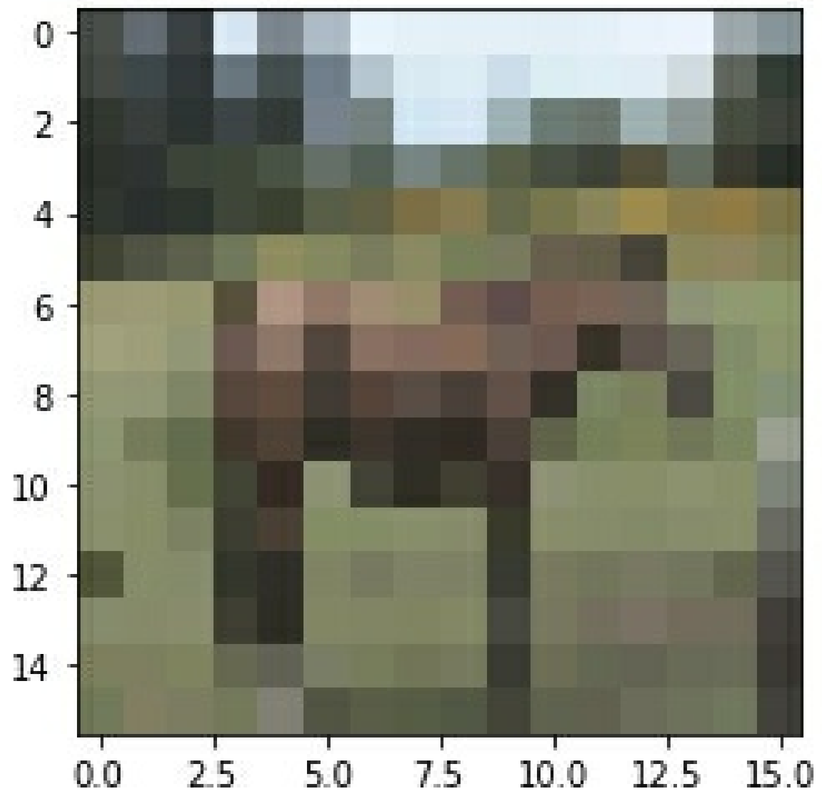
Output

```
[[ 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]]
(5, 5)
[[ 1  3  5]
[11 13 15]
[21 23 25]]
(3, 3)
```

Applying this one-skipping slicing to an image allows you to compress the image (and half the amount of information). (It's also half the amount of information - if you check the scale on the x-axis and y-axis, it's definitely 16 x 16 pixels.

```
plt.imshow(x_test[ 9999 ][:: 2 , :: 2 ])
```

Output:



Here is the output of the horse image ndarray, using all of numpy's basic aggregation functions.

```
photo = x_test[ 9999 ]
def print_labeled_val (label, val):
print( f"{label}:{val}" )

print_labeled_val( "Sum" , np.sum(photo))
print_labeled_val( "product of each element of arrays" ,np.prod(photo))
print_labeled_val( "mean" , np.mean(photo))
print_labeled_val( "standard deviation" , np.std(photo))
print_labeled_val( "variance" , np.var(photo))
print_labeled_val( "minimum value" , np.min(photo))
print_labeled_val( "maximum value" ,np.max(photo))
print_labeled_val( "the minimum element's index" , np.argmin(photo))
print_labeled_val( "the maximum element's index" , np.argmax(photo))
```

Output:

```
Sum:331662
product of each element of arrays:0
mean:107.962890625
standard deviation:46.36682798593862
variance:2149.8827374776206
minimum value:25
maximum value:252
the minimum element's index:1745
the maximum element's index:38
```

By writing ndarray[conditional expression], it is possible to apply the expression to all the elements of the array and create a new array containing the number of boolean elements of the original array.

```
z = np.array([ 1 , 2 , 3 , 4 , 5 ])
z <= 3
```

Output:

```
array([ True,  True, False, False, False])
```

Using this mechanism, it is possible to extract only those elements that meet the criteria as follows

```
z[z<= 3 ]
```

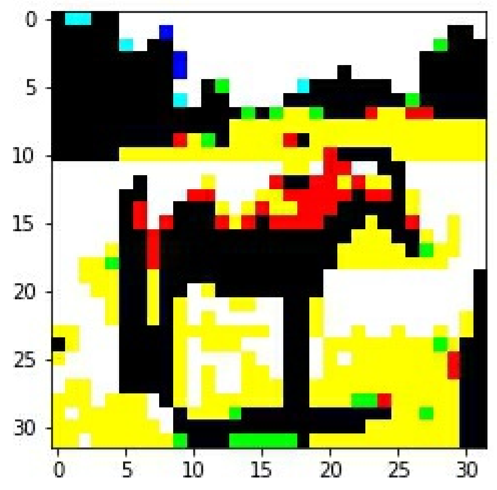
Output:

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

Using the ndarray conditional expression mechanism in image processing, it is possible to filter the RGB colors for an illustrative look, as shown in the following example

```
# RGBの各値が100を超えていたら255に変換, 100以下は0に変換する。
photo = x_test[ 9999 ]
photo_masked = np.where(photo > 100 , 255 , 0 )
plt.imshow(photo_masked)
```

Output:



Broadcast of Numpy.ndarray

Now let's take a look at one of the most important features in Numpy: broadcasts. Broadcast is a feature that automatically expands rows and columns when there are not enough elements. An obvious example is the ndarray and scalar quadrature. It is possible to perform calculations on all elements of ndarray, as shown below.

```
a = np.array([ 1 , 2 , 3 , 4 , 5 ])
print(a + 2 )
print(a - 2 )
print(a * 2 )
print(a / 2 )
```

Output:

```
[3 4 5 6 7]
[-1  0  1  2  3]
[ 2  4  6  8 10]
[0.5 1.  1.5 2.  2.5]
```

The broadcast can also be used to compute multidimensional arrays of different shapes as shown below. In the following quadratic operation of a_array and b_array, you can see that b_array is automatically expanded and computed as np.array([[10,10,10,10],[20,20,20]]).

```
a_array = np.array([[ 1 , 3 , 5 ],[ 7 , 9 , 11 ]])
b_array = np.array([[ 10 ],[ 20 ]])
print(a_array.shape)
print(b_array.shape)
```

Output:

```
(2, 3)
(2, 1)
```

```
print(a_array + b_array)
print(a_array - b_array)
print(a_array / b_array)
print(a_array * b_array)
```

Output:

```
[[11 13 15]
[27 29 31]]
[[ -9  -7  -5]
[-13 -11  -9]]
[[0.1  0.3  0.5 ]
[0.35 0.45 0.55]]
[[ 10  30  50]
[140 180 220]]
```

Calculating the inner product

One of the calculations that are often done using Numpy is the calculation of inner product. The inner product is represented by the following equation, which, when expanded, simply multiplies the elements of the w vector and the x vector in turn and adds them up, so it's not so difficult to calculate, but it is very useful for calculating ambiguous distances, such as for determining the similarity between images and words.

$$w \cdot x = \sum_{i=0}^{n} w_i x_i$$

There are three ways to write Numpy's inner product source code, as shown below, and the results are the same no matter which method you use, but if you're running in Python 3.6 or higher, I recommend using @ because it's the most readable and easiest to write.(As of 9/9/2020, Colab's Python was in the 3.6.9 version.)

```
w = np.array([ 1 , 2 , 3 ])
x = np.array([ 4 , 5 , 6 ]).T
print(np.dot(w,x))
print(w.dot(x))
print(w@x)
```

Output:

```
32
32
32
```

By the way, you can get the version of the Jupyter Notebook Python environment you are running using the magic command

```
!python --version
```

Output:

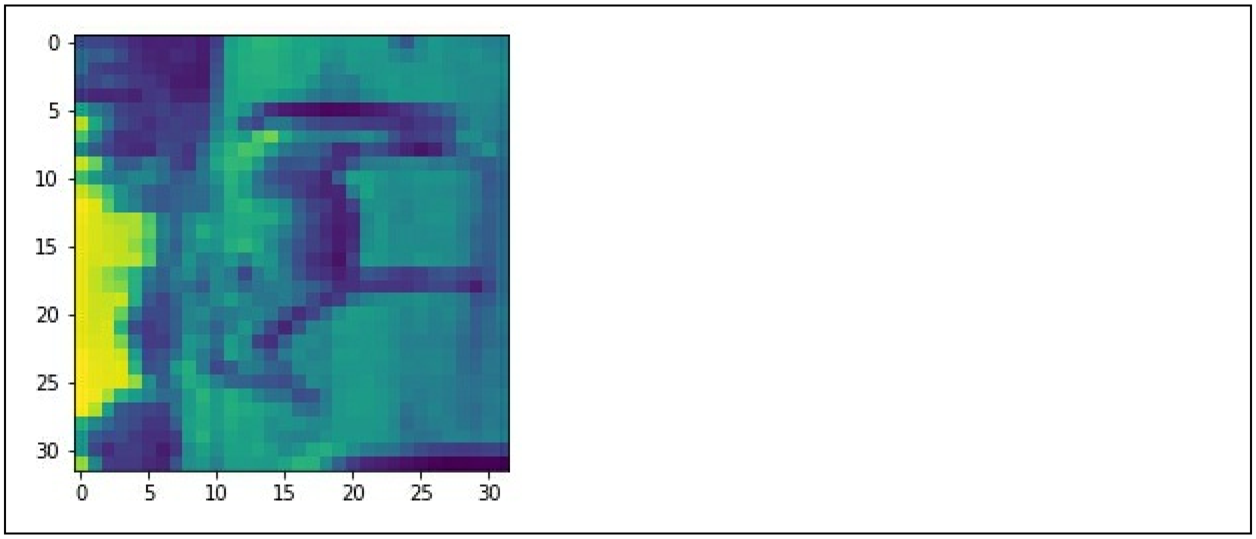
```
Python 3.6.9
```

T method in the previous inner-product calculation, where T stands for transpose, which means to replace the rows and columns. In calculating inner product, the shape of the object is very important: we can calculate the inner product of the L x N and N x M matrices, but not the N x L and N x M matrices. In the previous example, the product is automatically calculated as the product of a 1 x 3 shape w and a 3 x 1 x, without using the T method, because the product was calculated between two one-dimensional ndarrays.

If you apply this transposition matrix to the horse image mentioned above, the rows and columns are exchanged in the output as shown below, which should help you get an idea of what to expect.

```
plt.imshow(photo[:,:, 0 ].T)
```

Output:



Numpy has a lot of features that we haven't covered yet, but we recommend the Numpy excercises on Github, which will help you improve your Numpy skills by working through good questions.

11. Matplotlib

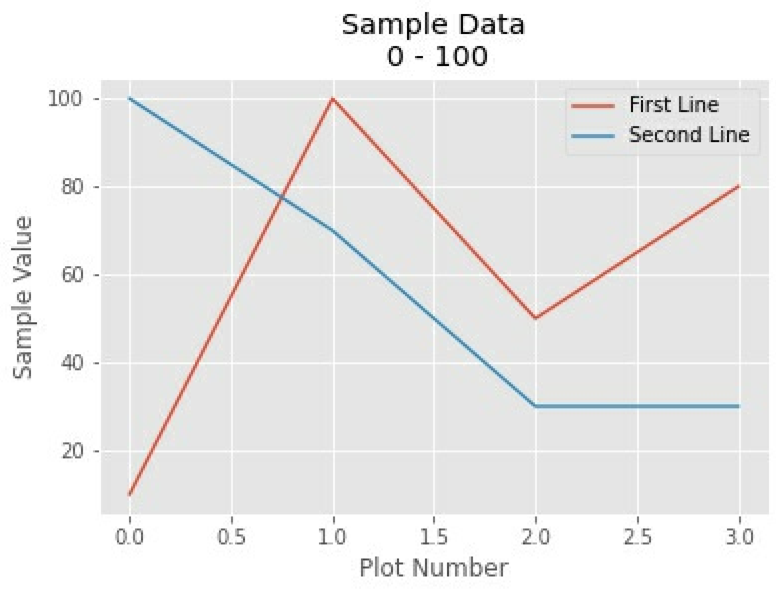
Matplotlib is one of the most popular data visualization tools for Python, and while Pandas also makes use of Matplotlib internally for easy graphing and display, this is only a small part of the functionality of Matplotlib. Matplotlib includes many modules, but this book cannot cover them all, so if you need them, please check the official website (https://matplotlib.org/py-modindex.html).

Legend, Title, X-axis and Y-axis labels

First, let's take a look at the most commonly used pyplot module, plot (commonly imported as import matplotlib.pyplot as plt and used in the abbreviated form plt). Run the following code to plot two data plt.plot whose x-axis is the index (an integer between 0 and 3) and whose y-axis is [10, 100, 50, 80], [100, 70, 30, 30], respectively. You can call the legend method to display the legend by passing the label keyword argument. You can pass label as a keyword argument and call the legend method to display the legend with the specified label. The x-axis label and y-axis label can be specified in the plt.xlabel and plt.ylabel methods, respectively. The whole graph title can handle line feed codes, so you don't need to use plt.subplot to display a two-row title.

```
import matplotlib.pyplot as plt
x = [ 0 , 1 , 2 , 3 ]
y = [ 10 , 100 , 50 , 80 ]
x2 = [ 0 , 1 , 2 , 3 ]
y2 = [ 100 , 70 , 30 , 30 ]

plt.plot(x, y, label= 'First Line' )
plt.plot(x2,y2, label= 'Second Line' )

plt.xlabel( 'Plot Number' )
plt.ylabel( 'Sample Value' )
plt.title( 'Sample Data\n 0 - 100' )
plt.legend()
plt.show()
```

Output:

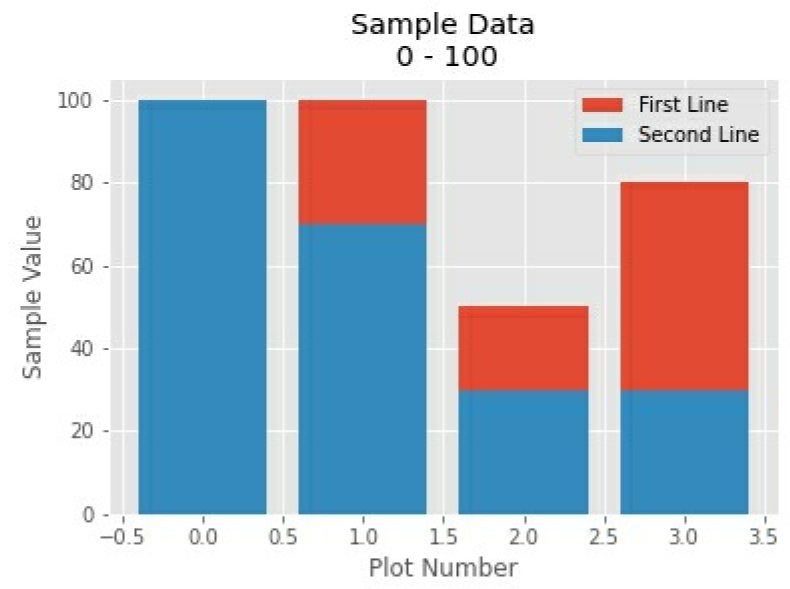


Bar Charts

Now let's view the data as a bar chart. The bar chart can be displayed using the pyplot.bar method.

```
import matplotlib.pyplot as plt
x = [ 0 , 1 , 2 , 3 ]
y = [ 10 , 100 , 50 , 80 ]
x2 = [ 0 , 1 , 2 , 3 ]
y2 = [ 100 , 70 , 30 , 30 ]

plt.bar(x, y, label= 'First Line' )
plt.bar(x2,y2, label= 'Second Line' )

plt.xlabel( 'Plot Number' )
plt.ylabel( 'Sample Value' )
plt.title( 'Sample Data\n 0 - 100' )
plt.legend()
plt.show()
```

Output:

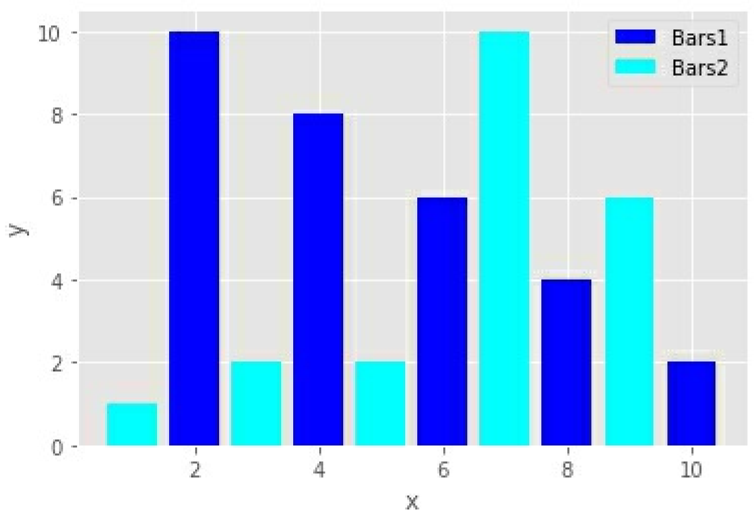


Series Color Specification

You can use color as an argument to the bar method to display it in any color you like, or you can use hex to specify RGB.

```
import matplotlib.pyplot as plt
import random
x = [i for i in range( 2 , 11 , 2 )]
y = [i for i in range( 10 , 1 , -2 )]

x2 = [i for i in range( 1 , 10 , 2 )]
y2 = [random.randint( 0 , 10 ) for i in range( 11 , 1 , -2 )]

plt.bar(x, y, label= 'Bars1' , color= 'blue' )
plt.bar(x2, y2, label= 'Bars2' , color= 'cyan' )

plt.xlabel( 'x' )
plt.ylabel( 'y' )
plt.title( '' )
plt.legend()
plt.show()
```

Output:

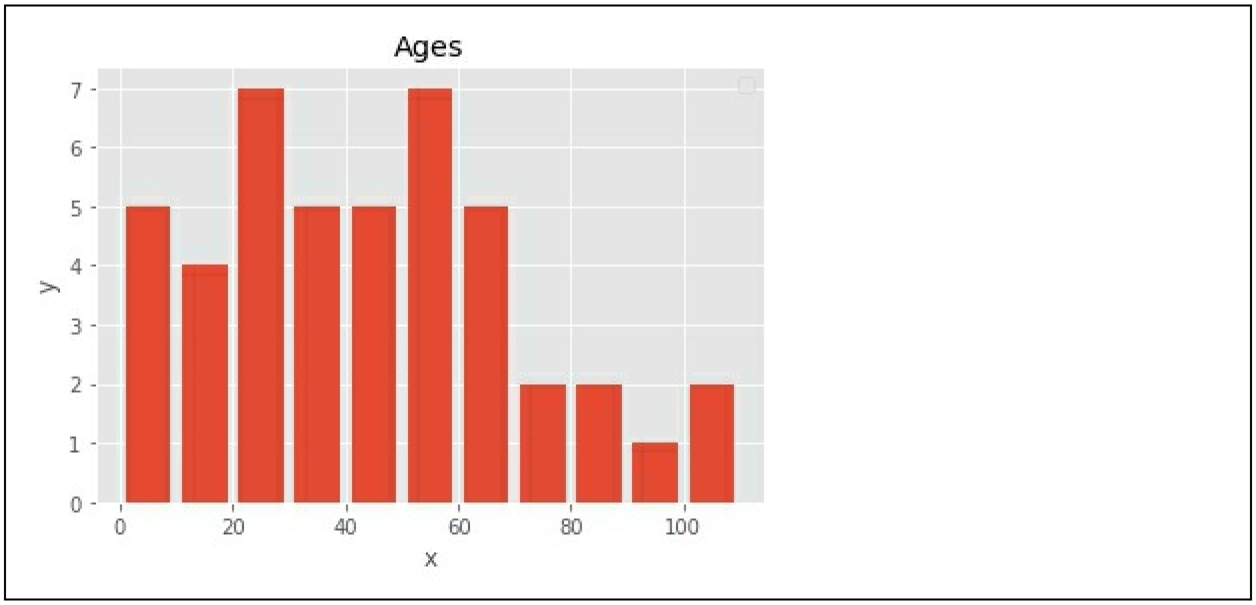


Histogram

Next, let's take a look at a histogram (frequency distribution) graph, which can be easily drawn using the plt.hist method. The first argument is the data and the second is the interval (the range of frequencies to be aggregated). In the following example, the range is specified by dividing by 10 from 0 to 110 using list comprehension. Rwidth specifies the width of each bar's interval, but I set it to 0.8 to make it easier to see (to prevent bars from sticking to each other).

```
import matplotlib.pyplot as plt

population_ages = [ 2 , 7 , 75 , 55 , 55 , 46 , 60 , 65 , 68 , 59 , 45 , 48 , 58 , 55
40 , 39 , 34 , 28 , 2 , 29 , 34 , 3 , 46 , 50 , 20 , 30 , 80 , 60 , 68 , 58 , 23 , 12 , 1
20 , 18 , 19 ]
bins = [i for i in range( 0 , 120 , 10 )]
plt.hist(population_ages, bins, histtype= 'bar' , rwidth= 0.8 )

plt.xlabel( 'x' )
plt.ylabel( 'y' )
plt.title( 'Ages' )
plt.legend()
plt.show()
```

Output:

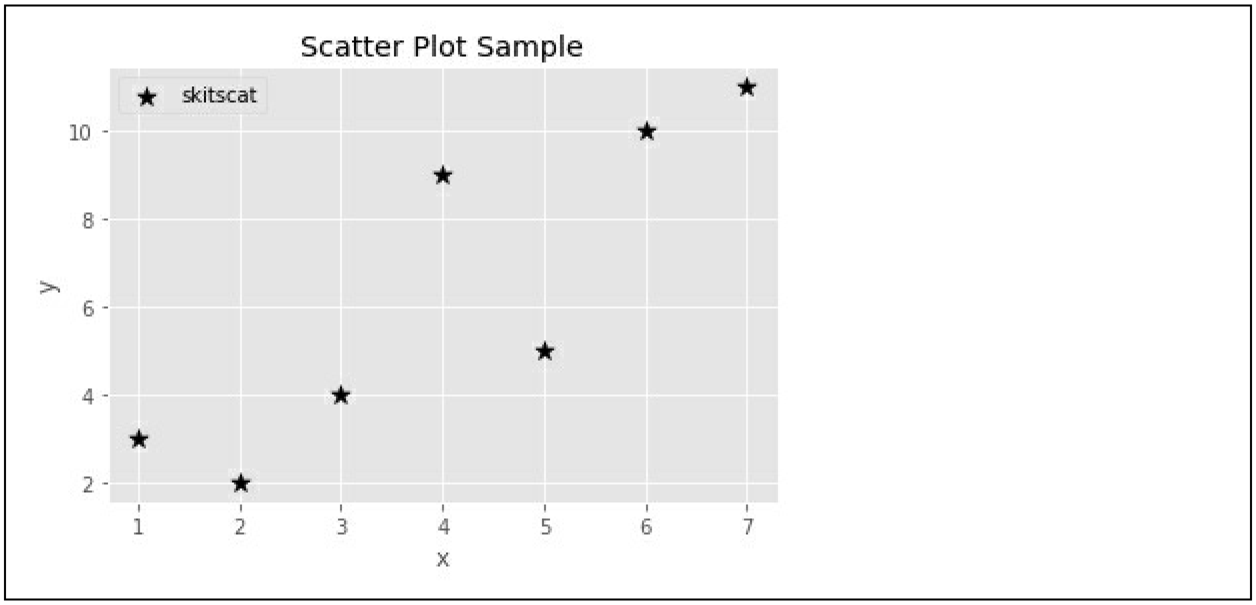


Scatter Plot

Scatter plots are useful when you want to see the correlation between two or more variables. plt.scatter can be used to draw a scatter plot. There are lots of other markers that allow you to specify a star, batten, and circle with "*", "x", and "o" respectively. Please refer to the official website (https://matplotlib.org/3.2.1/api/markers_api.html). If you want to increase the size of the marker, specify it with an s. The following sample shows a scatter plot with large black stars. The following sample shows a scatter diagram with large black stars.

```
x = [ 1 , 2 , 3 , 4 , 5 , 6 , 7 ]
y = [ 3 , 2 , 4 , 9 , 5 , 10 , 11 ]
plt.scatter(x, y, label= 'skitscat' , color= 'black' , marker= '*' , s= 100 )
plt.xlabel( 'x' )
plt.ylabel( 'y' )
plt.title( 'Scatter Plot Sample' )
plt.legend()
plt.show()
```

Output:



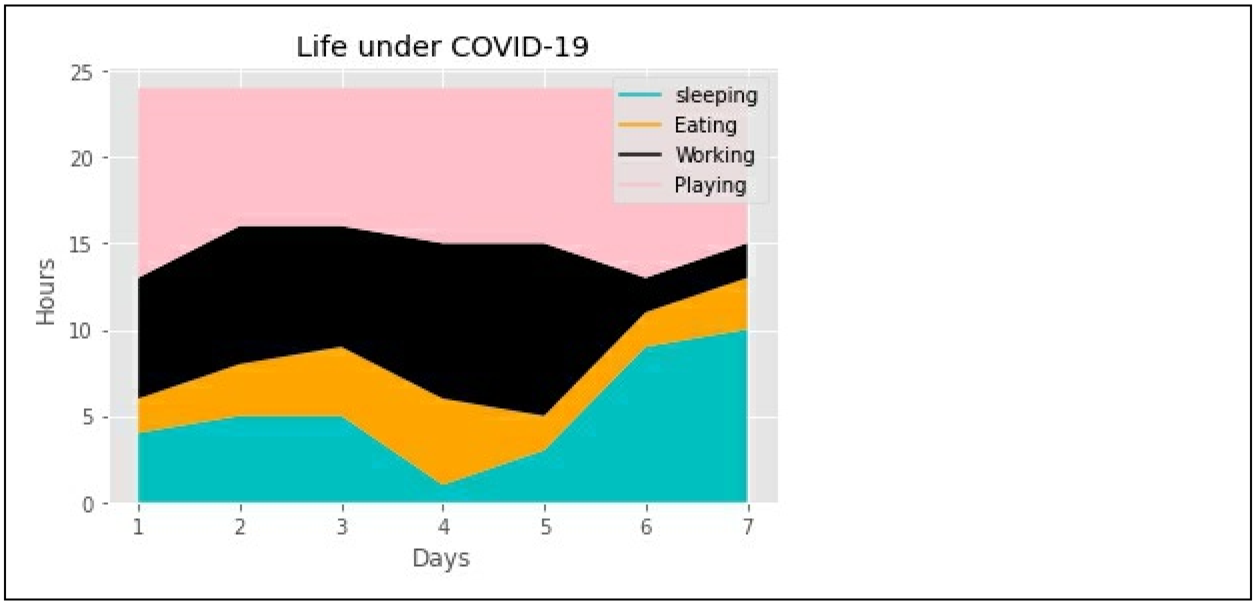
Stack Plot

Stack plots are used to show the percentage of data for each bar in a bar chart. Displaying the legend of the Stack Plot requires some technique, so please refer to the following code. In the age of the new pneumonia, when we have become a nationally reclusive population, it is very important to regulate the rhythm of our lives, so let's use plt.stackplot to visualize the percentage of time we sleep, eat, work and play in a week.

```
import matplotlib.pyplot as plt
days = [ 1 , 2 , 3 , 4 , 5 , 6 , 7 ]

sleeping = [ 4 , 5 , 5 , 1 , 3 , 9 , 10 ]
eating = [ 2 , 3 , 4 , 5 , 2 , 2 , 3 ]
working = [ 7 , 8 , 7 , 9 , 10 , 2 , 2 ]
playing = [ 11 , 8 , 8 , 9 , 9 , 11 , 9 ]
plt.plot([],[],color= 'c' , label= 'sleeping' )
plt.plot([],[],color= 'orange' ,label= 'Eating' )
plt.plot([],[],color= 'k' ,label= 'Working' )
plt.plot([],[],color= 'pink' ,label= 'Playing' )

plt.stackplot(days, sleeping, eating, working, playing, colors=[ 'c' , 'orange' , '
, 'pink' ])
```

```
plt.xlabel( 'Days' )
plt.ylabel( 'Hours' )
plt.title( 'Life under COVID-19' )
plt.legend()
plt.show()
```

Output:



Pie Charts

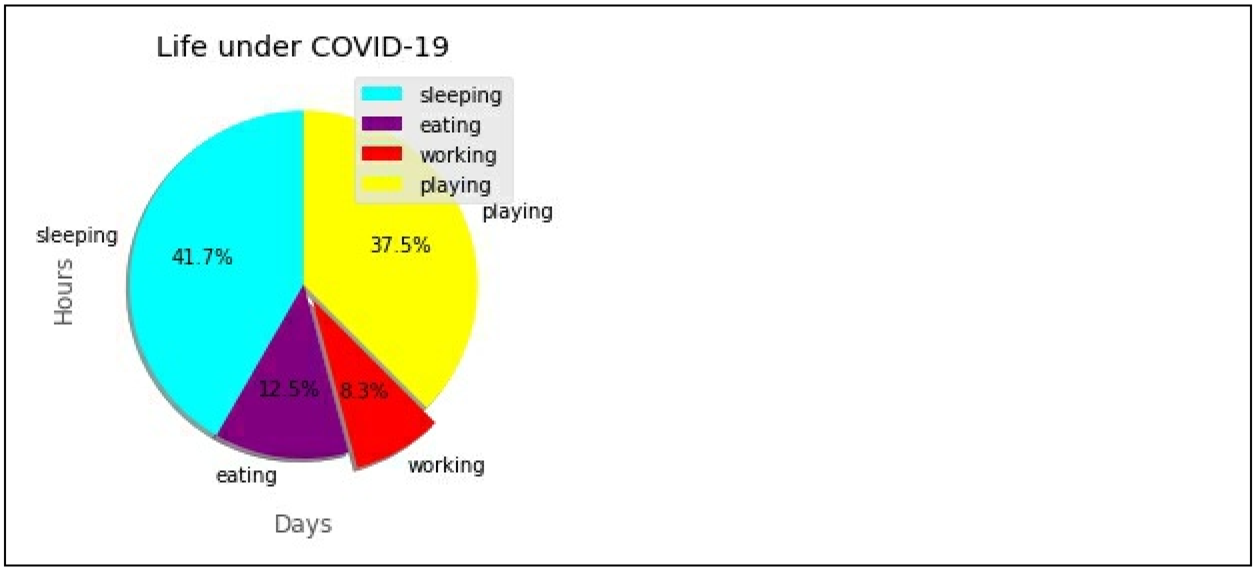
I think a useful way to show the percentage is pie charts (pie charts), and I expressed the percentage related to the activity time I just mentioned in pie charts as follows.

The startangle is used to tilt the pie chart at the angle at which it was started, and the shadow is used to create a shadow on the chart. I've also highlighted the third element (=playing) in explode so that it overhangs a bit; in autopct, I'm showing it as a percentage.

```
import matplotlib.pyplot as plt
days = [ 1 , 2 , 3 , 4 , 5 , 6 , 7 ]

sleeping = [ 4 , 5 , 5 , 1 , 3 , 9 , 10 ]
```

```
eating = [ 2 , 3 , 4 , 5 , 2 , 2 , 3 ]
working = [ 7 , 8 , 7 , 9 , 10 , 2 , 2 ]
playing = [ 11 , 8 , 8 , 9 , 9 , 11 , 9 ]
slices = [ 10 , 3 , 2 , 9 ]
activities = [ 'sleeping' , 'eating' , 'working' , 'playing' ]
colors = [ 'cyan' , 'purple' , 'red' , 'yellow' ]
plt.pie(slices, labels=activities, colors=colors, startangle= 90 , shadow= True
explode=( 0 , 0 , 0.1 , 0 ), autopct= "%1.1f%%" )

plt.xlabel( 'Days' )
plt.ylabel( 'Hours' )
plt.title( 'Life under COVID-19' )
plt.legend()
plt.show()
```

Output:



3D Graph

There are many phenomena in the world that cannot be judged in two dimensions. For example, it would not be possible to judge the risk of infection in Japan based solely on the number of people infected with a new type of coronavirus in the country. Other variables such as the number of PCR tests and fatality rates, as well as qualitative evaluations, should also be included in assessing the risk of infection.

Also, in deep learning supervised learning, the goal is to find weights that minimize the loss (the error between the correct answer and the model's output), but what may be a minuscule value in one dimension may converge (trap) to a saddle point that is a maximum value in another dimension.

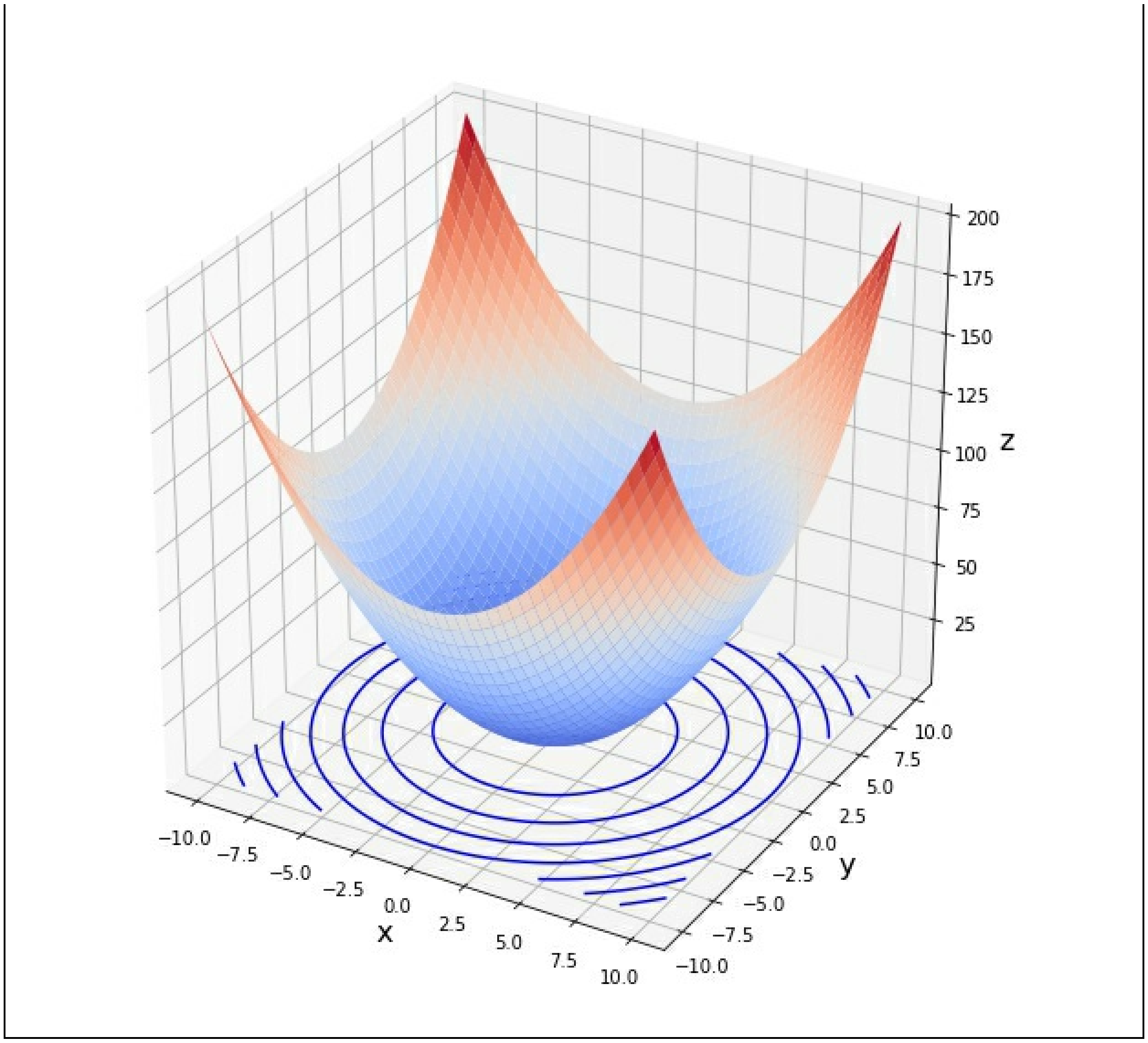
Visualizing up the dimensions is very important to capture the essence of things, so I will explain how to visualize it in three dimensions using Axes3D from mpl_toolkits.mplot3D.

Try running the following code as soon as possible.

```
# 3D Plot
import numpy as np
from matplotlib import cm
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

def plot_3d_func (X, Y, Z):
""" Three-dimensional plot of a function Z with inputs of two variables X
and Y


   Args:
     X: ndarray
     Y: ndarray
     Z: ndarray
"""
fig = plt.figure(figsize = ( 10 , 10 ))
ax = fig.add_subplot( 1 , 1 , 1 , projection= "3d" )

ax.set_xlabel( "x" , size = 16 )
ax.set_ylabel( "y" , size = 16 )
ax.set_zlabel( "z" , size = 16 )
ax.plot_surface(X, Y, Z, cmap=cm.coolwarm)
#  ax.plot_surface(X, Y, Z, cmap=cm.summer)

ax.contour(X, Y, Z, colors = "blue" , offset = -1 )
```

```
#  ax.scatter(X, Y, Z)

plt.show()

def func_z1 (X, Y):
return X** 2 + Y** 2

# Generates 256 elements equally spaced from -10 to 10
x = np.linspace( -10 , 10 , 256 )
y = np.linspace( -10 , 10 , 256 )

# Grid Generation
X, Y = np.meshgrid(x, y)

# Calling func_z1
Z = func_z1(X, Y)

# 3D Plot
plot_3d_func(X, Y, Z)
```

Ouput:



You may now see a contour graph with a bowl-shaped contour line at the bottom, as shown in the figure above. This is a graph that visualizes the following

$$Z = X^2 + Y^2$$

You can draw a three-dimensional surface by Axes3D.plot_surface and Axes3D.contour.

You can specify the colormap with cmap, which is the argument to plot_surface. The commented cm.summer also has beautiful colors, so please give it a try. Also, if you use Axes3D.scatter instead of contour, you can draw a 3D scatter diagram. In this example, we have drawn a function, but 3D scattergrams are also often used to plot real data.

Numpy.meshgrid is a method for generating lattices. You can think of a lattice grid as a grid of all combinations of elements in the domain of the x- and y-axes. It's hard to explain in words, so let's run the following source code.

```
x_sample = np.array([ 1 , 2 , 3 ])
y_sample = np.array([ 1 , 2 , 3 ])
X_sample, Y_sample = np.meshgrid(x_sample, y_sample)
print(X_sample)
print(Y_sample)
```

Output:

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

Above, we could easily get the grid points for all combinations of integers with x from 1 to 3 and y from 1 to 3. This is very useful for visualizing the output of a function that takes x and y as its arguments. In the previous 3D graph, we generated 256 elements on the x and y axes, equally spaced from -10 to 10, respectively, and generated all combinations as a grid sequence as well.

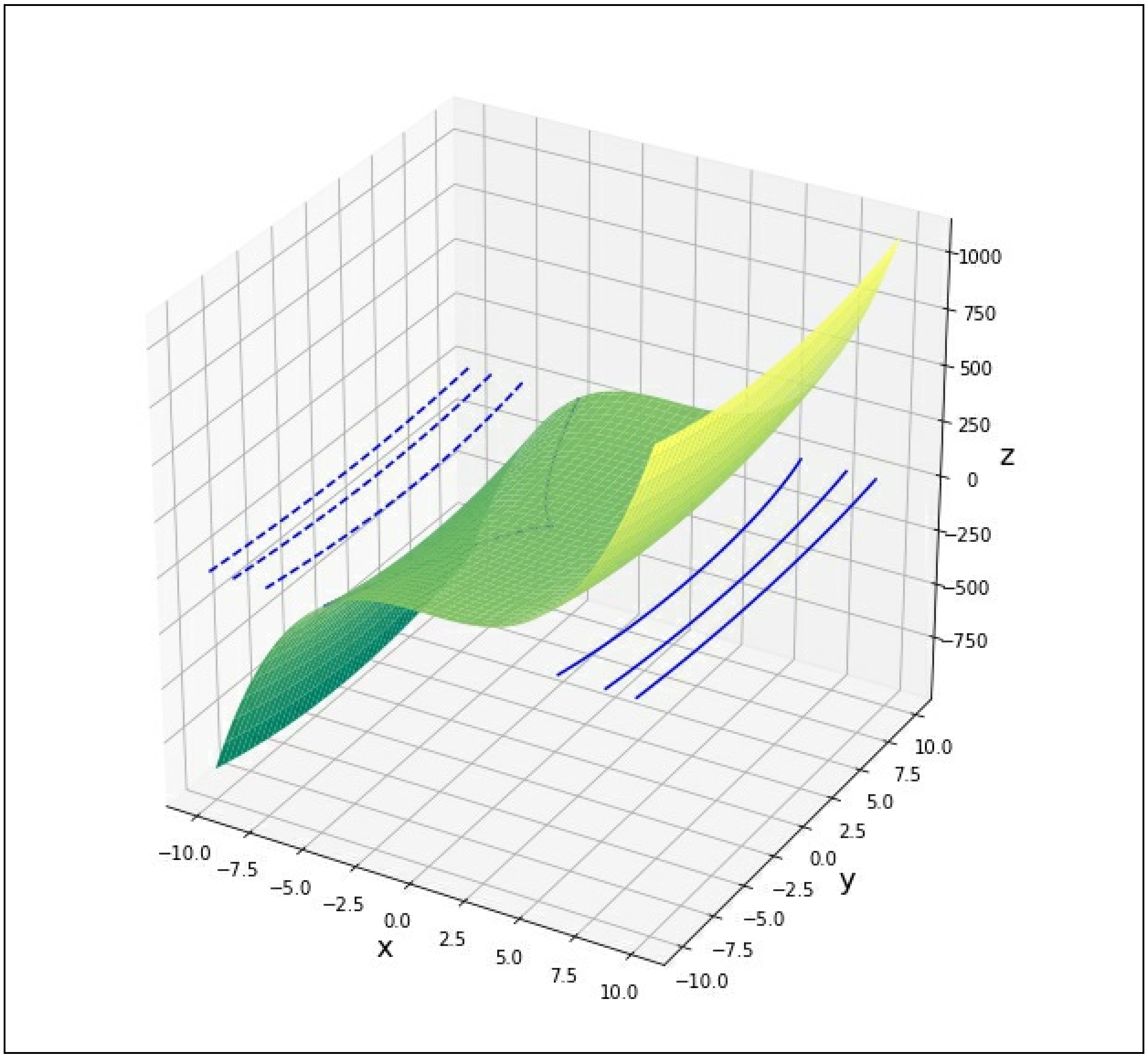
Then, let's display another function, $Z = X^3 + Y^2$.

If you comment out the cmap='summer' line in plot_3d_func and run the code, output is as follows.

```
def func_z2 (X, Y):
return X** 3 + Y** 2
```

```
Z = func_z2(X, Y)
plot_3d_func(X, Y, Z)
```

Output:

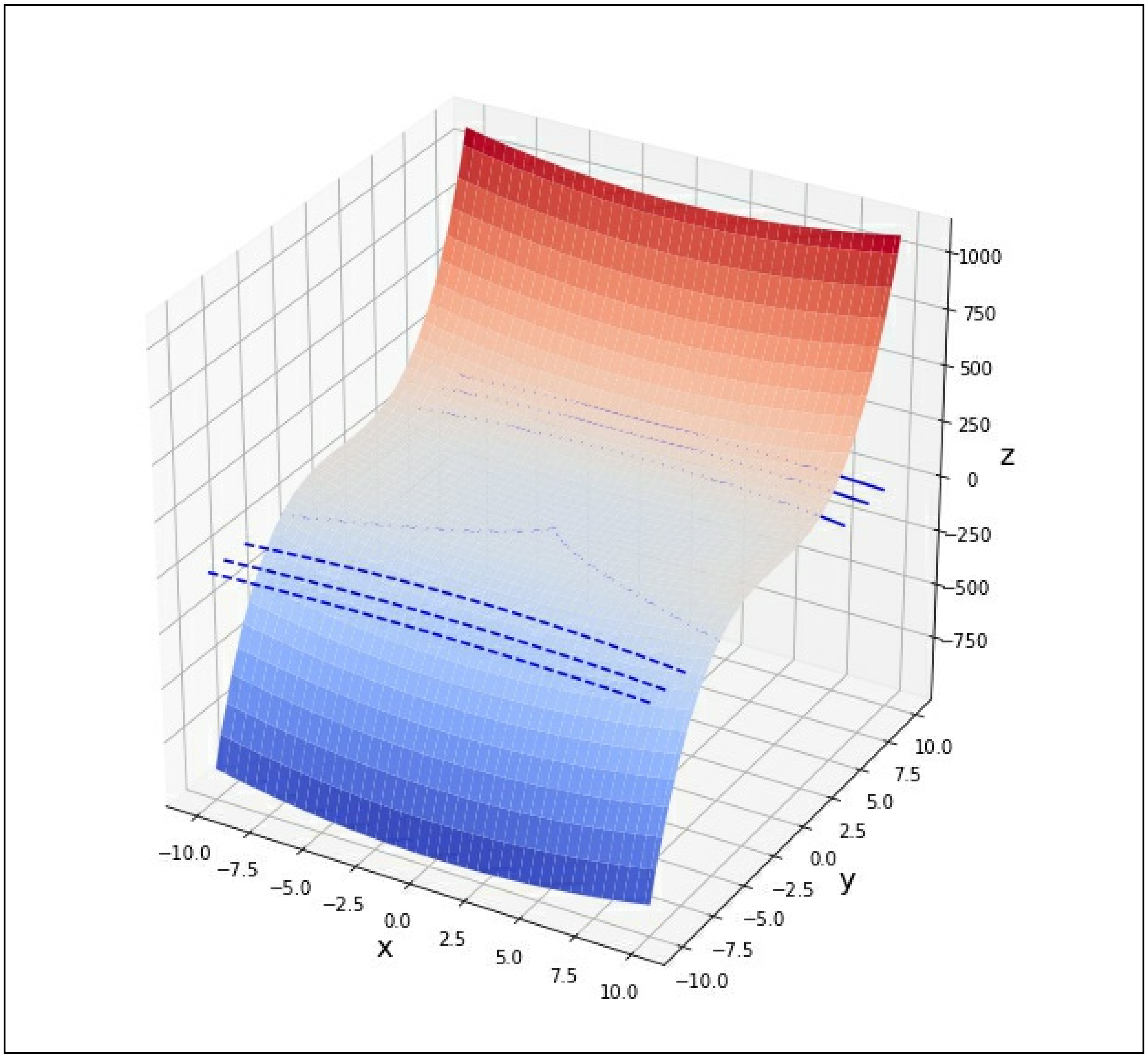


Then, let's display the graph of function, $Z = X^3 + Y^2$.
Again, I set the cmap to coolwarm.

```
def func_z3 (X, Y):
return X** 2 + Y** 3

Z = func_z3(X, Y)
```

```
plot_3d_func(X, Y, Z)
```

Output:



Please refer to the official mplot3d tutorial - Matplotlib 2.0.2 documentation to learn more about visualization techniques in three dimensions, as you can learn more diverse methods in the official mplot3d tutorial.

I hope that by the end of this chapter you have an idea of how powerful Matplotlib's visualization capabilities are, and that you can draw any kind of graph. There are some features that are not covered in this book, so please check the official tutorial (https://matplotlib.org/tutorials/introductory/pyplot.html) if you need it.

12. Scikit-Learn

Scikit-Learn provides a variety of machine learning algorithms that you can easily try out. If you are a beginner, there are three main points that you might get stuck with when using Scikit-Learn.

1. pre-processing of real data
2. selection of algorithms
3. selection of hyperparameters

Since you have been able to use Numpy/Pandas/Matplotlib in the previous chapter to freely transform and visualize the shape and dimensions of your data, I hope that you are able to solve the 1) pre-processing task. By running the samples in this chapter and trying them out, I hope you will see that you can easily build a machine learning model and solve the prediction task with a small amount of code.

2) For algorithm selection, Scikit-Learn's official website has a very useful diagram called Choosing the right estimator - scikit-learn 0.22.2 documentation, which can be found here , please refer to. The main point is that by answering the questions sequentially, such as how much data is available and whether it is a classification problem that applies a non-continuous label or a regression problem that predicts a continuous number, it is possible to decide which algorithm to use.

3) Regarding the selection of hyperparameters (parameters passed as execution arguments in each algorithm, number of training epochs, etc.), it is difficult even for professionals in machine learning for a living, but there are widely known methods such as random search, Bayesian optimization, grid search, etc., so if you are interested in these methods, please contact please search for these words to get to the point. As long as the algorithm used is not too wrong, it is possible to achieve a certain degree of practical accuracy by running simulations while looking at evaluation indicators such as loss and accuracy, so we won't go into details in this book.

In this chapter, you will first learn how to classify your sample data using

the SVM/RandomForest/XGBoost software bundled with Scikit-learn.
Finally, you will train your machine learning model on real data to perform classification (labeling) and regression (guessing).

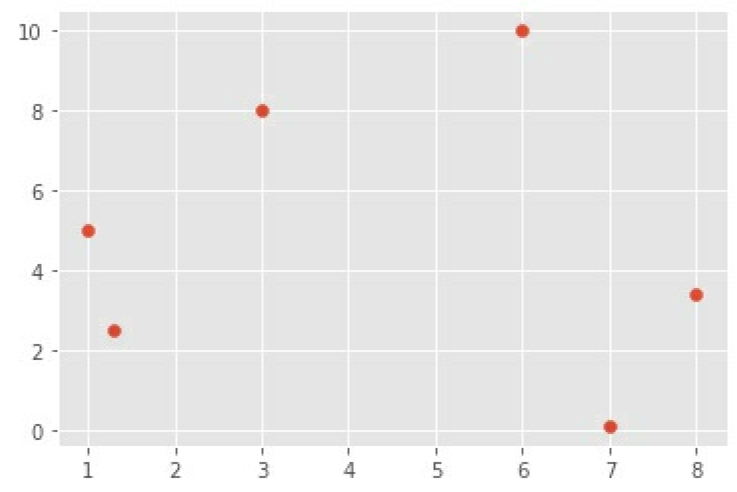
SVM:Support Vector Machine

SVM is a classification method backed by advanced mathematical theory, and until the ILSVRC in 2012, image recognition was often referred to as SVM, and the basic idea behind SVM is very clear: by finding the boundary plane (or boundary line in the case of two-dimensional data) that maximizes the distance between the data and classification. To do this, SVM takes the approach of mapping high-dimensional data and classifying it linearly in high-dimensional space, and this mapping is called a kernel function, and mapping is called a kernel trick I'll call it.

It's not easy to understand just by the explanation, so let's look at the sample data and the working code.
First, let's plot the data we want to classify on a scatter plot using matplotlib.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import style
style.use( "ggplot" )
from sklearn import svm
x = [ 1 , 3 , 8 , 1.3 , 7 , 6 ]
y = [ 5 , 8 , 3.4 , 2.5 , 0.1 , 10 ]
plt.scatter(x, y)
plt.show()
```

Output:



As we can't enter the list format into the Scikit-Learn model, we will convert it to a Numpy.ndarray format, which is a two-dimensional array. Also, since SVM is a supervised classification, we need to prepare some teacher labels. The teacher labels correspond to classification labels, where 0 is label 1 and 1 is label 2.

```
X = np.array([
   [ 1 , 5 ],
   [ 3 , 8 ],
   [ 8 , 3.4 ],
   [ 1.3 , 2.5 ],
   [ 7 , 0.1 ],
   [ 6 , 10 ]]
)

t = np.array([ 0 , 0 , 1 , 0 , 1 , 1 ])
```

The following code then defines the SVM classifier. The kernel function uses a linear map.

```
clf = svm.SVC(kernel= 'linear' )
```

The following code will actually do the machine learning(training) for you.

```
clf.fit(X, t)
```

Output:

```
SVC(C=1.0, break_ties=False, cache_size=200, class_weight=None,
coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='scale',
kernel='linear',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

Now that we've already trained the model, we'll try to solve the prediction task immediately.
Given a new point (0.38, 0.8), we want to try to predict which label this will be. From a human's perspective, it looks like it will be given the label 0 (i.e., the same label as (1, 5)), but I was able to classify it as expected when I predicted it with the SVM.

```
print(clf.predict(np.array([[ 0.38 , 0.8 ]])))
```

Output:

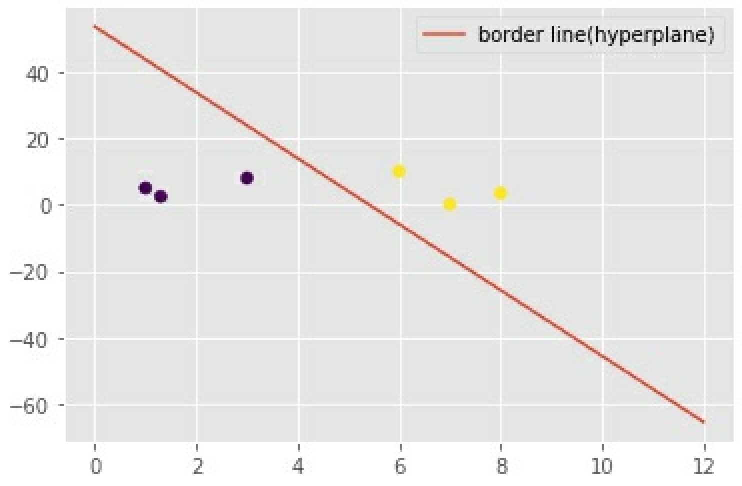
```
[0]
```

Let's take a look at the following code to visualize the boundaries that the trained SVM uses to make decisions.

```
w = clf.coef_[ 0 ]

a = -w[ 0 ] / w[ 1 ]

print( f"weight:{a}" )

xx = np.linspace( 0 , 12 )
yy = a * xx - clf.intercept_[ 0 ] / w[ 1 ]

h0 = plt.plot(xx, yy, '-' , label= "border line(hyperplane)" )

plt.scatter(X[:, 0 ], X[:, 1 ], c = y)
plt.legend()
plt.show()
```

Output:

```
weight:-9.899998505029213
```



Let's take a look at four other unknowns, run the prediction task, and calculate the percentage of correct answers (Accuracy). The accuracy is calculated as the percentage of all events for which the prediction agrees with the true result.

```
X_test = np.array([[ 0.38 , 0.8 ], [ 6 , 8 ], [ 7 , 7 ], [ 8 , 4 ], [ 2 , 2 ]])
y_test = np.array([ 0 , 1 , 1 , 1 , 0 ])
prediction = clf.predict(X_test)
print( f"Accuracy:{ 100 * len(y_test[y_test == prediction])/len(X_test)}%" )
```

Output:

```
Accuracy:100.0%
```

The predictions made in the SVM model we created in this study were 100% correct for no-tuning. I think this confirms that the kernel function is likely to be sufficiently accurate for linearly separable (linearly separable) sample data.

RandomForest

Next, let's create a model using a classification technique called

RandomForest, which applies a decision tree (DecisionTree), and then train and test it using the training data we just used.

```
#  Training by RandomForest
from sklearn.ensemble import RandomForestClassifier as classifier
random_forest = classifier(random_state= 0 )
random_forest.fit(X, t)

#  Inference by RandomForest
X_test = np.array([[ 0.38 , 0.8 ], [ 6 , 8 ], [ 7 , 7 ], [ 8 , 4 ], [ 2 , 2 ]])
y_test = np.array([ 0 , 1 , 1 , 1 , 0 ])
prediction = random_forest.predict(X_test)
print( f"Accuracy:{ 100 * len(y_test[y_test == prediction])/len(y_test)}%" )
```

Output:

```
Accuracy:100.0%
```

Even in the RandomForest model, the percentage of correct answers for sample data was 100% for no-tuning. The supervised learning method can ultimately be abstracted to the task of outputting a continuous or discontinuous pattern from the input data. So, the basic idea of decision trees is that if we check which features in the input data take what values in turn, and then create branching conditions (branching paths) like if statements in programming, we can eventually narrow the conditional refinement to a single output, which can be used for a prediction task. RandomForest randomly selects these features to create multiple decision trees, and then decides the final output of the model by majority vote based on the prediction results of these decision trees.

Learning with multiple models in this way is called ensemble learning. Also, like RandomForest, the method of training in parallel using multiple models with some data from the whole is called bagging.

XGBOOST(Classification)

On the other hand, the method of iteratively extracting some data and training it sequentially is called boosting. In the boosting method, a model is created first and trained, then parameters are adjusted to prioritize the correct answers to misrecognized data, then the wrong data is weighted against the wrong data in the previous model and the training proceeds, and finally the output is generated as a single model. This boosting is applied to a method called XgBoost, which is often seen at the top of the international data analysis contest Kaggle. Since boosting is sequential, the learning time tends to be more accurate than bagging, which can be learned in parallel, such as RandomForest, instead of taking more computational time. However, XgBoost is not known to be very accurate for typical linear separable problems, with a 40% correct rate for the following.

```
#  Training with the XgBoost Model
import xgboost as xgb
xgb_model = xgb.XGBClassifier(random_state= 0 )
xgb_model.fit(X, t)

#  Inference by XgBoost Model
X_test = np.array([[ 0.38 , 0.8 ], [ 6 , 8 ], [ 7 , 7 ], [ 8 , 4 ], [ 2 , 2 ]])
y_test = np.array([ 0 , 1 , 1 , 1 , 0 ])
prediction = xgb_model.predict(X_test)
print( f"Accuracy:{ 100 * len(y_test[y_test == prediction])/len(y_test)}%" )
```

Output

```
Accuracy:40.0%
```

XGBOOST(Regression)

So far, we have performed a classification task (predicting discontinuous labels) using a machine learning model that we did with SVM/RandomForest/XgBoost using sample data. Next, let's run a regression problem (predicting continuous values) in machine learning using XgBoostRegressor.

The sample training data is prepared for 10 people with the following height and weight data. The correct answer label is BMI. If we can derive the BMI function from the data alone, we will have trained a model to predict BMI from height and weight using machine learning.

```
X = np.array([
[ 160 , 45 ],
[ 180 , 70 ],
[ 190 , 65 ],
[ 155 , 48 ],
[ 168 , 65 ],
[ 140 , 25 ],
[ 95 , 15 ],
[ 70 , 8 ],
[ 200 , 100 ],
[ 178 , 66 ]
]
)

def get_bmi (hw):
""" Calculate BMI
   @Args
     hw(np.array([[height, width],...])): weight and height ndarray(ndim=2)
   @Returns
     bmi_arr(np.ndarray): BMI ndarray(ndim=1)
"""
return hw[:, 1 ]/((hw[:, 0 ]/ 100 )*(hw[:, 0 ]/ 100 ))

t = get_bmi(X)
```

The following code trains the XgBoostRegressor (the XgBoost model for the regression task). Also. We have prepared test data of four people's height and weight and performed a prediction task. Is it possible to predict the BMI of these four people?

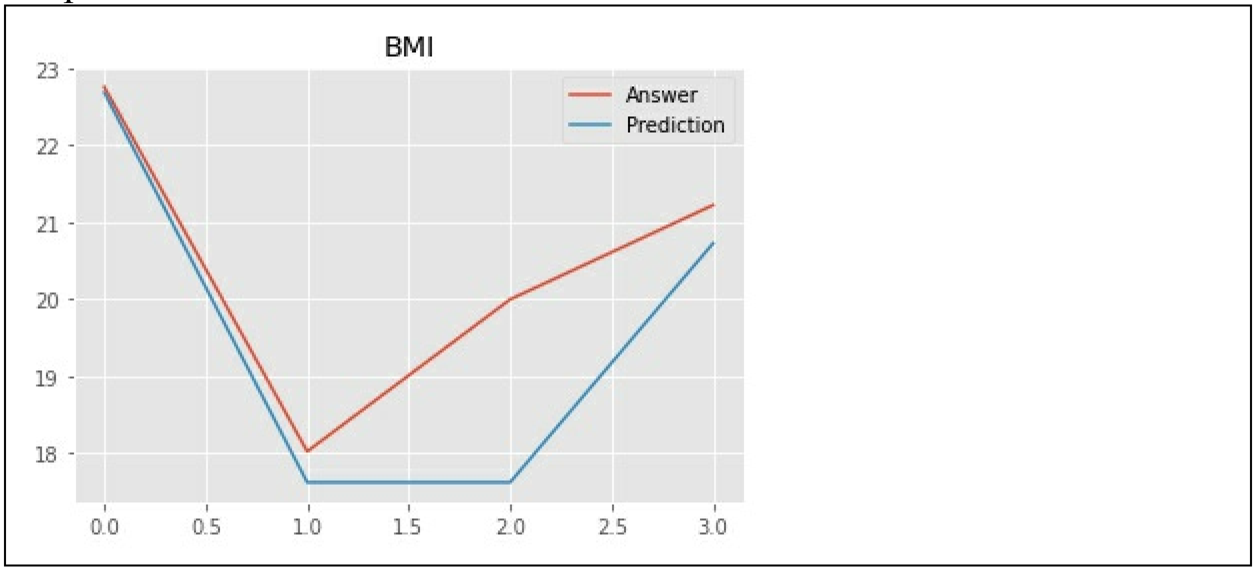
```
xgbr_model = xgb.XGBRegressor(random_state= 0 )
xgbr_model.fit(X, t)
```

```
X_test = np.array([
[ 169 , 65 ],
[ 158 , 45 ],
[ 150 , 45 ],
[ 175 , 65 ],
]
)
y_test = get_bmi(X_test)
prediction = xgbr_model.predict(X_test)
```

Let's visualize the correct and predicted results using matplotlib. Although there were only 10 training data, it can be seen that the correct answer and prediction results are quite close.

```
import matplotlib.pyplot as plt

plt.title( 'BMI' )
plt.plot(y_test, label= "Answer" )
plt.plot(prediction, label= "Prediction" )
plt.legend()


plt.show()
```

Output:



In the evaluation of the regression task, the error (the difference between the predicted data and the correct data) is used, the smaller the error, the more accurate the prediction is. There are various types of errors, but generally, the root mean square error (MSE) or the root mean square error (MSER) is used.

$$\mathrm{MSE}(c) = \frac{1}{n}\sum_{i=1}^{n}(x_i - c)^2$$

Scikit-Learn also has a handy method to calculate these values, so let's use it.

```
from sklearn.metrics import mean_squared_error
mse = mean_squared_error(y_test, prediction)
print( f"Root Mean Square:{np.sqrt(mse)}" )
```

Output

```
Root Mean Square:1.2304201718698495
```

Despite the fact that we have given only 10 data, and in addition, we have not done any tuning of the hyperparameters, we can see that the errors are relatively small.

13. Keras

Keras is a high-level library of neural networks developed by Google's Francois Chollet, who developed it as a wrapper for TensorFlow. It is an intuitive way to build neural networks, even for deep learning beginners.

LeNet

AlexNet, which was able to dominate the 2012 Image Recognition Contest (ILSVRC2012), is a type of convolutional neural network, and this convolutional neural network is based on LeNet, which was developed by Prof. Yann Lecun. Keras It is relatively easy to implement LeNet in LeNet, and we hope that you can experience the accuracy that we could not achieve with the simple neural network we used in the MNIST deep learning method at the beginning of this article.

First, we import the necessary libraries.

```
import os
import keras
from keras.models import Sequential
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import MaxPooling2D
from keras.layers.core import Activation
from keras.layers.core import Flatten, Dropout
from keras.layers.core import Dense
from keras.datasets import mnist
from keras.optimizers import Adam
from keras.callbacks import TensorBoard
```

Define the functions that define the LeNet model.

```
def lenet (input_shape, num_classes):

model =Sequential()
# Convolution using 20 5 x 5 filters and propagation of values to the next
layer by the activation function Relu

# When padding="same", the size of the feature map is 28 x 28, the same as
the input, so the output is 28 x 28 x 20.
model.add(Conv2D( 20 , kernel_size = 5 , padding= "same" ,
input_shape=input_shape, activation= "relu" ))
  # MaxPooling is performed to compress the feature map to 14 x 14 x 20
after processing to find the maximum value every 2 x 2 regions.
model.add(MaxPooling2D(pool_size=( 2 , 2 )))
  # A general technique that can be placed on CNN is to increase the number
of filters in the convolution the deeper it is.

model.add(Conv2D( 50 , kernel_size= 5 , padding= "same" , activation= "rel
))
  # Output: 7 x 7 x 50
model.add(MaxPooling2D(pool_size=( 2 , 2 )))
```

```
# The output is converted to a vector.
model.add(Flatten())
model.add(Dense( 500 , activation= "relu" ))

# It is used to classify num_classes.
model.add(Dense(num_classes))
model.add(Activation( "softmax" ))
return model
```

Load the MNIST data. Pre-processing is also useful if it is methodatized in one place, so we classify it as follows.

```
class MNISTDataset ():
def __init__ (self):
   self.image_shape = ( 28 , 28 , 1 )
   self.num_classes = 10

def get_batch (self):
   (x_train, y_train), (x_test, y_test) = mnist.load_data()

   x_train, x_test = [self.preprocess(d) for d in [x_train, x_test]]
   y_train, y_test = [self.preprocess(d, label_data= True ) for d in [y_train,
y_test]]
   return x_train, y_train, x_test, y_test

def preprocess (self, data, label_data=False):
    if label_data:
     data = keras.utils.to_categorical(data, self.num_classes)
    else :
     data = data.astype( "float32" )
     data /= 255
     shape = (data.shape[ 0 ],) + self.image_shape
     data = data.reshape(shape)
    return data
```

It is not easy to specify parameters for evaluation functions and so on every time, so we define a class to be trained by giving a model.

```
class Trainer ():
def __init__ (self, model, loss, optimizer):
   self._target = model
   self._target.compile(loss=loss, optimizer=optimizer, metrics=[ "accuracy"
])

def train (self, x_train, y_train, batch_size, epochs, validation_split):
     self._target.fit(
         x_train, y_train,
         batch_size=batch_size,
         epochs = epochs,
         validation_split = validation_split
     )
```

The MNIST data is loaded. Although normalization and shape transformation are done in the pre-processing, it is important to note that the input to the convolutional neural network is a 28 x 28 pixel image that is input as it is without flattening, so the shape is not the same as the input to the simple neural network of the shortest MNIST deep learning method at the beginning.

```
dataset = MNISTDataset()
x_train, y_train, x_test, y_test = dataset.get_batch()
print(x_train.shape)
print(y_train.shape)
print(x_test.shape)
print(y_test.shape)
```

Output:

```
(60000, 28, 28, 1)
(60000, 10)
(10000, 28, 28, 1)
(10000, 10)
```

The model is created by instantiating it as follows: summary() displays the summary of the model. Since the number of parameters is relatively large (1,256,080), it is recommended to use GPU mode of Colab because it will take a long time to train the model on CPU.

```
model = lenet(dataset.image_shape, dataset.num_classes)
model.summary()
```

Output:

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)              Output Shape           Param #
=================================================================
conv2d_9 (Conv2D)            (None, 28, 28, 20)        520
_________________________________________________________________
max_pooling2d_9 (MaxPooling2 (None, 14, 14, 20)        0
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 14, 14, 50)        25050
_________________________________________________________________
max_pooling2d_10 (MaxPooling (None, 7, 7, 50)          0
_________________________________________________________________
flatten_5 (Flatten)          (None, 2450)              0
_________________________________________________________________
dense_9 (Dense)              (None, 500)               1225500
_________________________________________________________________
dense_10 (Dense)             (None, 10)                5010
_________________________________________________________________
activation_5 (Activation)    (None, 10)                0
=================================================================
Total params: 1,256,080
Trainable params: 1,256,080
Non-trainable params: 0
_________________________________________________________________
```

Training is performed by the Trainer class. The learning rate is adjusted using Adam().

```
trainer = Trainer(model, loss= "categorical_crossentropy" ,
optimizer=Adam())
trainer.train(x_train, y_train, batch_size= 128 , epochs= 20 , validation_split
0.2 )
```

Output:

```
Train on 48000 samples, validate on 12000 samples
Epoch 1/20
48000/48000 [==============================] - 6s 118us/step -
loss: 0.0087 - accuracy: 0.9969 - val_loss: 0.0415 - val_accuracy: 0.9904
Epoch 2/20
48000/48000 [==============================] - 5s 111us/step -
loss: 0.0067 - accuracy: 0.9978 - val_loss: 0.0339 - val_accuracy: 0.9923
Epoch 3/20
48000/48000 [==============================] - 5s 110us/step -
loss: 0.0052 - accuracy: 0.9984 - val_loss: 0.0400 - val_accuracy: 0.9908
Epoch 4/20
48000/48000 [==============================] - 5s 109us/step -
loss: 0.0044 - accuracy: 0.9984 - val_loss: 0.0538 - val_accuracy: 0.9889
Epoch 5/20
48000/48000 [==============================] - 5s 109us/step -
loss: 0.0043 - accuracy: 0.9986 - val_loss: 0.0519 - val_accuracy: 0.9911
Epoch 6/20
48000/48000 [==============================] - 5s 109us/step -
loss: 0.0047 - accuracy: 0.9985 - val_loss: 0.0539 - val_accuracy: 0.9896
Epoch 7/20
48000/48000 [==============================] - 5s 109us/step -
loss: 0.0037 - accuracy: 0.9989 - val_loss: 0.0573 - val_accuracy: 0.9906
Epoch 8/20
48000/48000 [==============================] - 5s 109us/step -
loss: 0.0032 - accuracy: 0.9990 - val_loss: 0.0474 - val_accuracy: 0.9920
Epoch 9/20
48000/48000 [==============================] - 5s 109us/step -
loss: 0.0044 - accuracy: 0.9987 - val_loss: 0.0448 - val_accuracy: 0.9908
Epoch 10/20
48000/48000 [==============================] - 5s 109us/step -
```

```
loss: 0.0034 - accuracy: 0.9989 - val_loss: 0.0664 - val_accuracy: 0.9887
Epoch 11/20
48000/48000 [==============================] - 5s 109us/step -
loss: 0.0036 - accuracy: 0.9988 - val_loss: 0.0467 - val_accuracy: 0.9920
Epoch 12/20
48000/48000 [==============================] - 5s 109us/step -
loss: 0.0025 - accuracy: 0.9994 - val_loss: 0.0508 - val_accuracy: 0.9904
Epoch 13/20
48000/48000 [==============================] - 5s 109us/step -
loss: 0.0017 - accuracy: 0.9994 - val_loss: 0.0515 - val_accuracy: 0.9912
Epoch 14/20
48000/48000 [==============================] - 5s 109us/step -
loss: 0.0028 - accuracy: 0.9990 - val_loss: 0.0639 - val_accuracy: 0.9898
Epoch 15/20
48000/48000 [==============================] - 5s 109us/step -
loss: 0.0031 - accuracy: 0.9991 - val_loss: 0.0553 - val_accuracy: 0.9917
Epoch 16/20
48000/48000 [==============================] - 5s 109us/step -
loss: 0.0024 - accuracy: 0.9994 - val_loss: 0.0482 - val_accuracy: 0.9918
Epoch 17/20
48000/48000 [==============================] - 5s 110us/step -
loss: 5.0420e-04 - accuracy: 0.9998 - val_loss: 0.0547 - val_accuracy:
0.9919
Epoch 18/20
48000/48000 [==============================] - 5s 110us/step -
loss: 1.6923e-04 - accuracy: 0.9999 - val_loss: 0.0563 - val_accuracy:
0.9915
Epoch 19/20
48000/48000 [==============================] - 5s 109us/step -
loss: 0.0059 - accuracy: 0.9981 - val_loss: 0.0553 - val_accuracy: 0.9904
Epoch 20/20
48000/48000 [==============================] - 5s 109us/step -
loss: 0.0022 - accuracy: 0.9994 - val_loss: 0.0607 - val_accuracy: 0.9904
```

We will perform a prediction task for number recognition on the model we created and look at the loss and correctness rates. In the case of the neural network with flat input introduced in the chapter on the shortest MNIST at the beginning of this article, the percentage of correct answers in the test data was around 97-98%, while LeNet was able to increase the percentage of correct answers in the test data to 99.1%. They were able to improve the accuracy to the point where they were able to get one wrong or wrong out of 100 cards despite the fact that they were included. This feat is thought to be related to the fact that the structure of the convolutional network was inspired by the human visual cortex.

```
score = model.evaluate(x_test, y_test, verbose= 0 )
y_pred = model.predict(x_test)
print( f"Loss:{score[ 0 ]}, Accuracy:{score[ 1 ]}" )
```

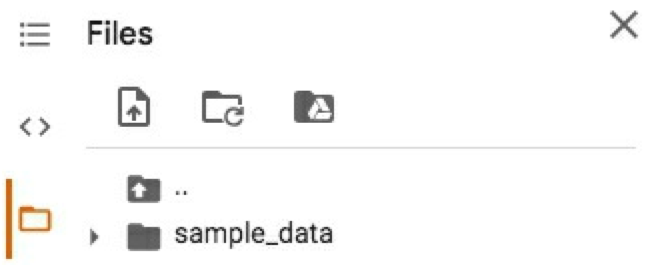
Output:

```
Loss:0.04733146464541562, Accuracy:0.9912999868392944
```

VGG

A pre-trained model of a data set of over a million images called ImageNet has been published using Oxford's VGG, which came in second place at ILSVRC2014. In this chapter, in order to develop a more practical artificial intelligence application, we will use this model to recognize an arbitrary image and run the source code to predict what is depicted in the image. First, we upload the image we want to recognize in the temporary area of Colab.

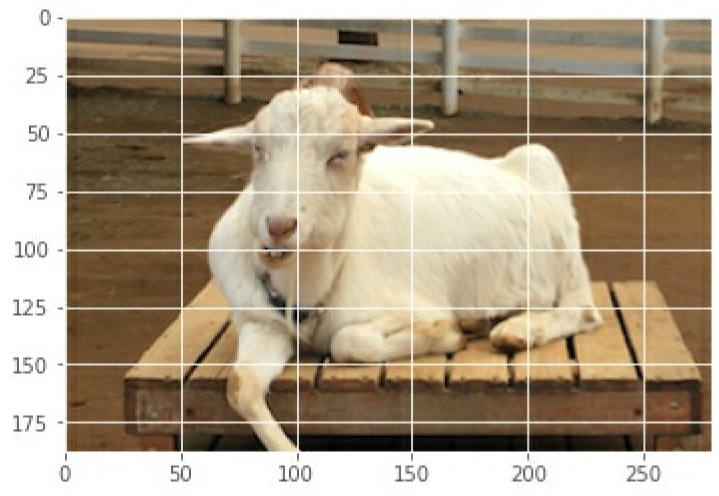
Click on the folder icon in the upper left menu of Colab, then click the upload button and upload any image file you want. If you put them in the most parent folder as shown below, you can refer to them by specifying the relative path of the current directory. The author has uploaded a goat image taken at a certain zoo in Japan.



First, let's make sure that the image is uploaded and handled by Colab using matplotlib.

```
import matplotlib.pyplot as plt
# Please upload any file from the left folder mark and then specify it.
image_path = "goat.png"
image = Image.load_img(image_path, target_size=( 188 , 280 ))
plt.imshow(image)
plt.show()
```

Output:



We then import the libraries needed to load the VGG pre-training model.

```
from keras.applications.vgg16 import VGG16
from keras.applications.vgg16 import preprocess_input, decode_predictions
import keras.preprocessing.image as Image
import numpy as np
```

The following simple code can be used to load a VGG16 model that has been pre-trained on ImageNet.

```
model = VGG16(weights= "imagenet" , include_top= True )
```

Next, let's try image recognition. The ImageNet is a dataset of more than a million 224 x 224 pixels images, so we resize the original images, train them, and run the prediction task. So, we use decode_predictions to convert it to a label.

```
image_path = "goat.png"
image = Image.load_img(image_path, target_size=( 224 , 224 ))

x = Image.img_to_array(image)
x = np.expand_dims(x, axis= 0 )
x = preprocess_input(x)

result = model.predict(x)
result = decode_predictions(result, top= 10 )[ 0 ]
for res in result:
print( f"Label:{res[ 1 ]}, Probability:{round( 100 *res[ 2 ], 1 )}%" )
```

Output:

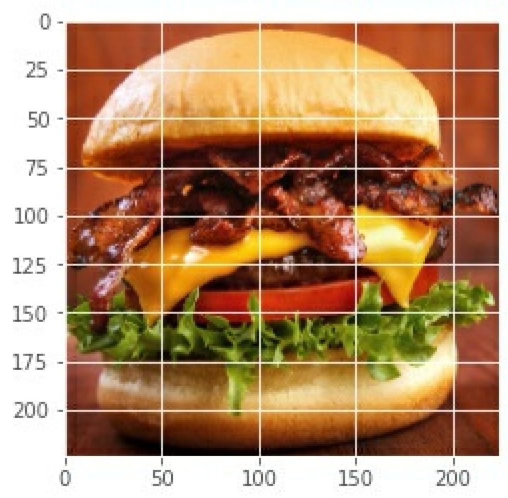
```
Label:ram, Probability:36.3%
Label:ox, Probability:17.8%
Label:borzoi, Probability:17.5%
Label:oxcart, Probability:7.1%
Label:Ibizan_hound, Probability:4.8%
Label:worm_fence, Probability:2.9%
Label:wallaby, Probability:2.8%
Label:llama, Probability:1.6%
Label:hog, Probability:1.0%
Label:kelpie, Probability:0.6%
```

The top 1 was a ram. It may be true, but there were no goats (goats) in the top 10 because there were not many Japanese goats in ImageNet's training data. It is well known that Japanese data and ImageNet data are different, so udon and duke noodles are judged as spaghetti. In order to handle Japanese image data, it is necessary to re-train a part of the model that is close to the output layer of the pre-trained model, called fine tuning. I won't cover fine tuning in this book, but if you're interested, there's an article in Qiita that looks at creating a face recognition AI by connecting all the coupled layers to VGG16 and using the GPU to fine-tune VGG16.

Since ImageNet is an American-centric dataset to begin with, I figured that if it looks American, it should be recognized, so I got an image of a hamburger and let it recognize it.

```
image_path = "hamburger2.png"
# Please upload any file from the left folder mark and then specify it.
image = Image.load_img(image_path, target_size=( 224 , 224 ))
plt.imshow(image)
plt.show()
```

Output:



```
x = Image.img_to_array(image)
x = np.expand_dims(x, axis= 0 )
x = preprocess_input(x)
```

```
result = model.predict(x)
result = decode_predictions(result, top= 10 )[ 0 ]

for res in result:
print( f"Label:{res[ 1 ]}, Probability:{round( 100 *res[ 2 ], 1 )}%" )
```

Output:

```
Label:cheeseburger, Probability:100.0%
Label:bagel, Probability:0.0%
Label:guacamole, Probability:0.0%
Label:hotdog, Probability:0.0%
Label:meat_loaf, Probability:0.0%
Label:potpie, Probability:0.0%
Label:burrito, Probability:0.0%
Label:plate, Probability:0.0%
Label:broccoli, Probability:0.0%
Label:mushroom, Probability:0.0%
```

Then, to my surprise, they answered with 100% confidence that it was 100% cheeseburgers, including the type of burger, and they were 100% correct. After all, it looks like we should treat ImageNet as an American eye.

Data link for the new Corona (covid-19)

A variety of data, including data on the number of tests and series of cases of the new coronaviruses by region and the number of people infected by the new coronaviruses, are available to the public, and various competitions are being held, so I've included links to each of the most famous national and international competitions, one for each.

If you are a data scientist, machine learning engineer, or other professional, using your data analysis skills to share the results of your analysis may result in saving lives. And if you're a beginner, if you've read this book carefully while running the source code, you'll be skilled enough to create predictive models and submit the results in a contest.Kaggle will also be able to use Numpy/Pandas/Matplotlib to create EDA (Many notebooks of the results of the exploratory data analysis) are already available, so you don't have to enter the contest to refer to them, which is very helpful. We hope you'll give it a shot!

Kaggle(English)

https://www.kaggle.com/covid19

SIGNATE(English/Japanese)

https://signate.jp/competitions/260