**Using Flask to make the model presentable web application.**

**What is Flask Python**

Flask is a web framework, it’s a Python module that lets you develop web applications easily. It’s has a small and easy-to-extend core: it’s a microframework that doesn’t include an ORM (Object Relational Manager) or such features.

It does have many cool features like url routing, template engine. It is a WSGI web app framework.

**Related course:** [Python Flask: Create Web Apps with Flask](https://gum.co/IMzBy)

**What is a Web Framework?**

A Web Application Framework or a simply a Web Framework represents a collection of libraries and modules that enable web application developers to write applications without worrying about low-level details such as protocol, thread management, and so on.

**What is Flask?**

Flask is a web application framework written in Python. It was developed by Armin Ronacher, who led a team of international Python enthusiasts called Poocco. Flask is based on the Werkzeg WSGI toolkit and the Jinja2 template engine.Both are Pocco projects.

**WSGI**

The Web Server Gateway Interface (Web Server Gateway Interface, WSGI) has been used as a standard for Python web application development. WSGI is the specification of a common interface between web servers and web applications.

**Werkzeug**

Werkzeug is a WSGI toolkit that implements requests, response objects, and utility functions. This enables a web frame to be built on it. The Flask framework uses Werkzeg as one of its bases.

**jinja2**

jinja2 is a popular template engine for Python.A web template system combines a template with a specific data source to render a dynamic web page.

This allows you to pass Python variables into HTML templates like this:

<html>

<head>

<title>{{ title }}</title>

</head>

<body>

<h1>Hello {{ username }}</h1>

</body>

</html>

**Microframework**

Flask is often referred to as a microframework. It is designed to keep the core of the application simple and scalable.

Instead of an abstraction layer for database support, Flask supports extensions to add such capabilities to the application.

**Why is Flask a good web framework choice?**

Unlike the Django framework, Flask is very Pythonic. It’s easy to get started with Flask, because it doesn’t have a huge learning curve.

On top of that it’s very explicit, which increases readability. To create the “Hello World” app, you only need a few lines of code.

This is a boilerplate code example.

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route('/')

def hello\_world():

return 'Hello World!'

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

If you want to develop on your local computer, you can do so easily. Save this program as server.py and run it with python server.py.

$ python server.py

\* Serving Flask app "hello"

\* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)

It then starts a web server which is available only on your computer. In a web browser open localhost on port 5000 (the url) and you’ll see “Hello World” show up.

To host and develop online, you can use PythonAnywhere

Some example output:

flask

It’s a microframework, but that doesn’t mean your whole app should be inside one single Python file. You can and should use many files for larger programs, to handle complexity.

Micro means that the Flask framework is simple but extensible. You may all the decisions: which database to use, do you want an ORM etc, Flask doesn’t decide for you.

Flask is one of the most popular web frameworks, meaning it’s up-to-date and modern. You can easily extend it’s functionality. You can scale it up for complex applications.

PIL – Python Image Library

The **Python Imaging Library** adds image processing capabilities to your Python interpreter.

This library provides extensive file format support, an efficient internal representation, and fairly powerful image processing capabilities.

The core image library is designed for fast access to data stored in a few basic pixel formats. It should provide a solid foundation for a general image processing tool.

Let’s look at a few possible uses of this library.

**Image Archives**

The Python Imaging Library is ideal for image archival and batch processing applications. You can use the library to create thumbnails, convert between file formats, print images, etc.

The current version identifies and reads a large number of formats. Write support is intentionally restricted to the most commonly used interchange and presentation formats.

**Image Display**

The current release includes Tk PhotoImage and BitmapImage interfaces, as well as a Windows DIB interface that can be used with PythonWin and other Windows-based toolkits. Many other GUI toolkits come with some kind of PIL support.

For debugging, there’s also a show() method which saves an image to disk, and calls an external display utility.

**Image Processing**

The library contains basic image processing functionality, including point operations, filtering with a set of built-in convolution kernels, and colour space conversions.

The library also supports image resizing, rotation and arbitrary affine transforms.

There’s a histogram method allowing you to pull some statistics out of an image. This can be used for automatic contrast enhancement, and for global statistical analysis.

TORCHVISION

**What is torchvision?**

Torchvision is a library for Computer Vision that goes hand in hand with PyTorch. It has utilities for efficient Image and Video transformations, some commonly used pre-trained models, and some datasets ( torchvision does not come bundled with PyTorch, you will have to install it separately. )

This library is part of the [PyTorch](http://pytorch.org/) project. PyTorch is an open source machine learning framework.

Features described in this documentation are classified by release status:

*Stable:* These features will be maintained long-term and there should generally be no major performance limitations or gaps in documentation. We also expect to maintain backwards compatibility (although breaking changes can happen and notice will be given one release ahead of time).

*Beta:* Features are tagged as Beta because the API may change based on user feedback, because the performance needs to improve, or because coverage across operators is not yet complete. For Beta features, we are committing to seeing the feature through to the Stable classification. We are not, however, committing to backwards compatibility.

*Prototype:* These features are typically not available as part of binary distributions like PyPI or Conda, except sometimes behind run-time flags, and are at an early stage for feedback and testing.

The [torchvision](https://pytorch.org/vision/stable/index.html#module-torchvision) package consists of popular datasets, model architectures, and common image transformations for computer vision.

TRANSFORMING AND AUGMENTING IMAGES

Transforms are common image transformations available in the torchvision.transforms module. They can be chained together using [Compose](https://pytorch.org/vision/stable/generated/torchvision.transforms.Compose.html#torchvision.transforms.Compose). Most transform classes have a function equivalent: [functional transforms](https://pytorch.org/vision/stable/transforms.html#functional-transforms) give fine-grained control over the transformations. This is useful if you have to build a more complex transformation pipeline (e.g. in the case of segmentation tasks).

Most transformations accept both [PIL](https://pillow.readthedocs.io/) images and tensor images, although some transformations are [PIL-only](https://pytorch.org/vision/stable/transforms.html#transforms-pil-only) and some are [tensor-only](https://pytorch.org/vision/stable/transforms.html#transforms-tensor-only). The [Conversion Transforms](https://pytorch.org/vision/stable/transforms.html#conversion-transforms) may be used to convert to and from PIL images.

The transformations that accept tensor images also accept batches of tensor images. A Tensor Image is a tensor with (C, H, W) shape, where C is a number of channels, H and W are image height and width. A batch of Tensor Images is a tensor of (B, C, H, W) shape, where B is a number of images in the batch.

The expected range of the values of a tensor image is implicitly defined by the tensor dtype. Tensor images with a float dtype are expected to have values in [0, 1). Tensor images with an integer dtype are expected to have values in [0, MAX\_DTYPE] where MAX\_DTYPE is the largest value that can be represented in that dtype.

Randomized transformations will apply the same transformation to all the images of a given batch, but they will produce different transformations across calls. For reproducible transformations across calls, you may use [functional transforms](https://pytorch.org/vision/stable/transforms.html#functional-transforms).

The following examples illustrate the use of the available transforms:

* [Illustration of transforms](https://pytorch.org/vision/stable/auto_examples/plot_transforms.html" \l "sphx-glr-auto-examples-plot-transforms-py)
* Inserting image...
* [Tensor transforms and JIT](https://pytorch.org/vision/stable/auto_examples/plot_scripted_tensor_transforms.html#sphx-glr-auto-examples-plot-scripted-tensor-transforms-py)
* Inserting image...

MODELS AND PRE-TRAINED WEIGHTS

The torchvision.models subpackage contains definitions of models for addressing different tasks, including: image classification, pixelwise semantic segmentation, object detection, instance segmentation, person keypoint detection, video classification, and optical flow.

General information on pre-trained weights

TorchVision offers pre-trained weights for every provided architecture, using the PyTorch [torch.hub](https://pytorch.org/docs/stable/hub.html#module-torch.hub). Instancing a pre-trained model will download its weights to a cache directory. This directory can be set using the *TORCH\_HOME* environment variable. See [torch.hub.load\_state\_dict\_from\_url()](https://pytorch.org/docs/stable/hub.html#torch.hub.load_state_dict_from_url) for details.

Using the pre-trained models

Before using the pre-trained models, one must preprocess the image (resize with right resolution/interpolation, apply inference transforms, rescale the values etc). There is no standard way to do this as it depends on how a given model was trained. It can vary across model families, variants or even weight versions. Using the correct preprocessing method is critical and failing to do so may lead to decreased accuracy or incorrect outputs.

All the necessary information for the inference transforms of each pre-trained model is provided on its weights documentation. To simplify inference, TorchVision bundles the necessary preprocessing transforms into each model weight.

**What is so good with torchvision ?**

* Since it is an accompaniment to PyTorch, it automatically comes with the **GPU support**. ( **So, it is FAST !** )
* Its development philosophy is to be simple in implementation ( eg: without an extensive argument set for its functions ) . The developers have kept it separately from PyTorch to keep it lean and lightweight.
* It is **ready to use** ! ( comes with sample data-set ( CIFAR10, CelebA etc) , some commonly used pre-trained models (ResNet18, maskRCNN\_resnet50 etc ) and even sample starter codes for some of the typical AI/ Machine Learning Problems ( Image Classification, Semantic Segmentation , Keypoint detection etc ) — all inbuilt into its library
* It is developed and maintained by the Facebook AI team, and supported by the python community.

**1. Datasets —**

torchvision comes with an option to readily download a set of most commonly used datasets — more than enough to get you started to most of your implementations . It downloads the datasets onto your local system ( first time only ) and then from here on — you can directly use that in your program by referencing it .

Some of the supported datasets are —

* CIFAR
* CelebA
* COCO
* Omniglot
* VOC
* Flickr
* FashionMNIST
* …. and much more .

**2. Pre-trained Models**

To facilitate transfer learning, torch vision has specific pre-trained models for Image classification , Object Detection , Instance segmentation and even Video Classification. It downloads the models on to you local system ( first time only ).

some of these pre-trained models available are —

* ResNet 3D 18 ( video classification )
* MASK R-CNN ( Instance Segmentation )
* AlexNet , VGG, ResNet, Inception etc ( for typical Image classification problems )

Here is a comparison on different models for G-ops v/s accuracy



**3. Transforms**

Transforms are easily callable modules within the library that are used for Image Augmentations and transformations . The illustration below shows 3 of the sample transforms —



some sample transforms in torchvision ( Image by Author)

**Some of the other common/ important transforms are**

* ToTensor() — Convert anImage datasets to Tensors
* CenterCrop() — Crops with the center fixed
* Normalize() — Normalize the pixel values to that of the dataset that you are using
* Pad() — to give a padding or border
* … and many more

**4. IO —**

This module consists of functions for efficient reading and writing of Video and Image Inputs . ( More details are covered in Part 2 )

This is a good module to use if you are planning to do some Image transformations, without having to use the dataloader / dataset / transforms module .

**5.OPS —**

This module contains very specific operators for computer-vision ( built on native C++ ) and with python bindings.

eg: functions specifically used for Mask R-CNN like ROI ( Region of Interest )align operators. We will go deeper with an example in the later parts.

**6.References / Training Scripts**

This module contains end-to-end sample training scripts for implementing some of the standard AI problems — eg: image classification

object detection , key point detection , instance segmentation etc.

You can get started on solving some of the sample AI problems using torchvision using their tutorial .

**Convolutional Neural Network**

By now, you might already know about machine learning and deep learning, a computer science branch that studies the design of algorithms that can learn. Deep learning is a subfield of machine learning that is inspired by artificial neural networks, which in turn are inspired by biological neural networks.

A specific kind of such a deep neural network is the convolutional network, which is commonly referred to as CNN or ConvNet. It's a deep, feed-forward artificial neural network. **Remember** that feed-forward neural networks are also called multi-layer perceptrons(MLPs), which are the quintessential deep learning models. The models are called "feed-forward" because information flows right through the model. There are no feedback connections in which outputs of the model are fed back into itself.

CNNs specifically are inspired by the biological visual cortex. The cortex has small regions of cells that are sensitive to the specific areas of the visual field. This idea was expanded by a captivating experiment done by Hubel and Wiesel in 1962 (if you want to know more, here's a [**video**](https://www.youtube.com/watch?v=Cw5PKV9Rj3o)). In this experiment, the researchers showed that some individual neurons in the brain activated or fired only in the presence of edges of a particular orientation like vertical or horizontal edges. For example, some neurons fired when exposed to vertical sides and some when shown a horizontal edge. Hubel and Wiesel found that all of these neurons were well ordered in a columnar fashion and that together they were able to produce visual perception. This idea of specialized components inside of a system having specific tasks is one that machines use as well and one that you can also find back in CNNs.

Convolutional neural networks have been one of the most influential innovations in the field of computer vision. They have performed a lot better than traditional computer vision and have produced state-of-the-art results. These neural networks have proven to be successful in many different real-life case studies and applications, like:

* Image classification, object detection, segmentation, face recognition;
* Self driving cars that leverage CNN based vision systems;
* Classification of crystal structure using a convolutional neural network;
* And many more, of course!

To understand this success, you'll have to go back to 2012, the year in which Alex Krizhevsky used convolutional neural networks to win that year's ImageNet Competition, reducing the classification error from 26% to 15%.

**Note** that ImageNet Large Scale Visual Recognition Challenge (ILSVRC) began in the year 2010 is an annual competition where research teams assess their algorithms on the given data set and compete to achieve higher accuracy on several visual recognition tasks.

This was the time when neural networks [**regained**](https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf) prominence after quite some time. This is often called the "third wave of neural networks". The other two waves were in the 1940s until the 1960s and in the 1970s to 1980s.

Alright, you know that you'll be working with feed-forward networks that are inspired by the biological visual cortex, but what does that actually mean?

Take a look at the picture below:

Convolutional neural network

The image shows you that you feed an image as an input to the network, which goes through multiple convolutions, subsampling, a fully connected layer and finally outputs something.

But what are all these concepts?

1. The convolution layer computes the output of neurons that are connected to local regions or receptive fields in the input, each computing a dot product between their weights and a small receptive field to which they are connected to in the input volume. Each computation leads to extraction of a feature map from the input image. In other words, imagine you have an image represented as a 5x5 matrix of values, and you take a 3x3 matrix and slide that 3x3 window or kernel around the image. At each position of that matrix, you multiply the values of your 3x3 window by the values in the image that are currently being covered by the window. As a result, you'll get a single number that represents all the values in that window of the images. You use this layer to filtering: as the window moves over the image, you check for patterns in that section of the image. This works because of filters, which are multiplied by the values outputted by the convolution.

1. The objective of subsampling is to get an input representation by reducing its dimensions, which helps in reducing overfitting. One of the techniques of subsampling is max pooling. With this technique, you select the highest pixel value from a region depending on its size. In other words, max pooling takes the largest value from the window of the image currently covered by the kernel. For example, you can have a max-pooling layer of size 2 x 2 will select the maximum pixel intensity value from 2 x 2 region. You're right to think that the pooling layer then works a lot like the convolution layer! You also take a kernel or a window and move it over the image; The only difference is the function that is applied to the kernel and the image window isn't linear.
2. Max pooling
4. The objective of the fully connected layer is to flatten the high-level features that are learned by convolutional layers and combining all the features. It passes the flattened output to the output layer where you use a softmax classifier or a sigmoid to predict the input class label.

For more information, you can go [**here**](http://cs231n.github.io/convolutional-networks).

**The Fashion-MNIST Data Set**

Before you go ahead and load in the data, it's good to take a look at what you'll exactly be working with! The [**Fashion-MNIST**](https://arxiv.org/abs/1708.07747) dataset is a dataset of Zalando's article images, with 28x28 grayscale images of 70,000 fashion products from 10 categories, and 7,000 images per category. The training set has 60,000 images, and the test set has 10,000 images. You can double check this later when you have loaded in your data! ;)

Fashion-MNIST is similar to the MNIST dataset that you might already know, which you use to classify handwritten digits. That means that the image dimensions, training and test splits are similar to the MNIST dataset. **Tip**: if you want to learn how to implement an Multi-Layer Perceptron (MLP) for classification tasks with this latter dataset, go to [**this tutorial**](https://www.datacamp.com/tutorial/deep-learning-python).

You can find the Fashion-MNIST dataset [**here**](https://github.com/zalandoresearch/fashion-mnist), but you can also load it with the help of specific TensorFlow and Keras modules. You'll see how this works in the next section!

**Load the Data**

Keras comes with a library called datasets, which you can use to load datasets out of the box: you download the data from the server and speeds up the process since you no longer have to download the data to your computer. The train and test images along with the labels are loaded and stored in variables train\_X, train\_Y, test\_X, test\_Y, respectively.

from keras.datasets import fashion\_mnist

(train\_X,train\_Y), (test\_X,test\_Y) = fashion\_mnist.load\_data()

**Analyze the Data**

Let's now analyze how images in the dataset look like. Even though you know the dimension of the images by now, it's still worth the effort to analyze it programmatically: you might have to rescale the image pixels and resize the images.

import numpy as np

from keras.utils import to\_categorical

import matplotlib.pyplot as plt

%matplotlib inline

print('Training data shape : ', train\_X.shape, train\_Y.shape)

print('Testing data shape : ', test\_X.shape, test\_Y.shape)

('Training data shape : ', (60000, 28, 28), (60000,))

('Testing data shape : ', (10000, 28, 28), (10000,))

From the above output, you can see that the training data has a shape of 60000 x 28 x 28 since there are 60,000 training samples each of 28 x 28 dimension. Similarly, the test data has a shape of 10000 x 28 x 28 since there are 10,000 testing samples.

Blurry ankle boots

The output of above two plots looks like an ankle boot, and this class is assigned a class label of 9. Similarly, other fashion products will have different labels, but similar products will have same labels. This means that all the 7,000 ankle boot images will have a class label of 9.

**Data Preprocessing**

As you could see in the above plot, the images are grayscale images have pixel values that range from 0 to 255. Also, these images have a dimension of 28 x 28. As a result, you'll need to preprocess the data before you feed it into the model.

* As a first step, convert each 28 x 28 image of the train and test set into a matrix of size 28 x 28 x 1 which is fed into the network.

train\_X = train\_X.reshape(-1, 28,28, 1)

test\_X = test\_X.reshape(-1, 28,28, 1)

train\_X.shape, test\_X.shape

((60000, 28, 28, 1), (10000, 28, 28, 1))

In one-hot encoding, you convert the categorical data into a vector of numbers. The reason why you convert the categorical data in one hot encoding is that machine learning algorithms cannot work with categorical data directly. You generate one boolean column for each category or class. Only one of these columns could take on the value 1 for each sample. Hence, the term one-hot encoding.

For your problem statement, the one hot encoding will be a row vector, and for each image, it will have a dimension of 1 x 10. The important thing to note here is that the vector consists of all zeros except for the class that it represents, and for that, it is 1. For example, the ankle boot image that you plotted above has a label of 9, so for all the ankle boot images, the one hot encoding vector would be [0 0 0 0 0 0 0 0 1 0].

**The Network**

The images are of size 28 x 28. You convert the image matrix to an array, rescale it between 0 and 1, reshape it so that it's of size 28 x 28 x 1, and feed this as an input to the network.

You'll use three convolutional layers:

* The first layer will have 32-3 x 3 filters,
* The second layer will have 64-3 x 3 filters and
* The third layer will have 128-3 x 3 filters.

In addition, there are three max-pooling layers each of size 2 x 2.

Architecture of the model

**Neural Network Architecture**

In Keras, you can just stack up layers by adding the desired layer one by one. That's exactly what you'll do here: you'll first add a first convolutional layer with Conv2D(). Note that you use this function because you're working with images! Next, you add the Leaky ReLU activation function which helps the network learn non-linear decision boundaries. Since you have ten different classes, you'll need a non-linear decision boundary that could separate these ten classes which are not linearly separable.

More specifically, you add Leaky ReLUs because they attempt to fix the problem of dying Rectified Linear Units (ReLUs). The ReLU activation function is used a lot in neural network architectures and more specifically in convolutional networks, where it has proven to be more effective than the widely used logistic sigmoid function. As of 2017, this activation function is the most popular one for deep neural networks. The ReLU function allows the activation to be thresholded at zero. However, during the training, ReLU units can "die". This can happen when a large gradient flows through a ReLU neuron: it can cause the weights to update in such a way that the neuron will never activate on any data point again. If this happens, then the gradient flowing through the unit will forever be zero from that point on. Leaky ReLUs attempt to solve this: the function will not be zero but will instead have a small negative slope.

Next, you'll add the max-pooling layer with MaxPooling2D() and so on. The last layer is a Dense layer that has a softmax activation function with 10 units, which is needed for this multi-class classification problem.

**Compile the Model**

After the model is created, you compile it using the Adam optimizer, one of the most popular optimization algorithms. You can read more about this optimizer [**here**](https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning). Additionally, you specify the loss type which is categorical cross entropy which is used for multi-class classification, you can also use binary cross-entropy as the loss function. Lastly, you specify the metrics as accuracy which you want to analyze while the model is training.

**Train the Model**

It's finally time to train the model with Keras' fit() function! The model trains for 20 epochs. The fit() function will return a history object; By storying the result of this function in fashion\_train, you can use it later to plot the accuracy and loss function plots between training and validation which will help you to analyze your model's performance visually.

However, it looks like the model is overfitting, as the validation loss is 0.4396 and the validation accuracy is 92%. Overfitting gives an intuition that the network has memorized the training data very well but is not guaranteed to work on unseen data, and that is why there is a difference in the training and validation accuracy.

You probably need to handle this. In next sections, you'll learn how you can make your model perform much better by adding a Dropout layer into the network and keeping all the other layers unchanged.

But first, let's evaluate the performance of your model on the test set before you come on to a conclusion.

Training and validation accuracy graph

Training and validation loss graph

From the above two plots, you can see that the validation accuracy almost became stagnant after 4-5 epochs and rarely increased at certain epochs. In the beginning, the validation accuracy was linearly increasing with loss, but then it did not increase much.

The validation loss shows that this is the sign of overfitting, similar to validation accuracy it linearly decreased but after 4-5 epochs, it started to increase. This means that the model tried to memorize the data and succeeded.

With this in mind, it's time to introduce some dropout into our model and see if it helps in reducing overfitting.

Numpy

What is NumPy?

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

At the core of the NumPy package, is the *ndarray* object. This encapsulates *n*-dimensional arrays of homogeneous data types, with many operations being performed in compiled code for performance. There are several important differences between NumPy arrays and the standard Python sequences:

* NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically). Changing the size of an *ndarray* will create a new array and delete the original.
* The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory. The exception: one can have arrays of (Python, including NumPy) objects, thereby allowing for arrays of different sized elements.
* NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python’s built-in sequences.
* A growing plethora of scientific and mathematical Python-based packages are using NumPy arrays; though these typically support Python-sequence input, they convert such input to NumPy arrays prior to processing, and they often output NumPy arrays. In other words, in order to efficiently use much (perhaps even most) of today’s scientific/mathematical Python-based software, just knowing how to use Python’s built-in sequence types is insufficient - one also needs to know how to use NumPy arrays.

The points about sequence size and speed are particularly important in scientific computing. As a simple example, consider the case of multiplying each element in a 1-D sequence with the corresponding element in another sequence of the same length. If the data are stored in two Python lists, a and b, we could iterate over each element:

c **=** **[]**

**for** i **in** range**(**len**(**a**)):**

c**.**append**(**a**[**i**]\***b**[**i**])**

This produces the correct answer, but if a and b each contain millions of numbers, we will pay the price for the inefficiencies of looping in Python. We could accomplish the same task much more quickly in C by writing (for clarity we neglect variable declarations and initializations, memory allocation, etc.)

**for** **(**i **=** **0;** i **<** rows**;** i**++)** **{**

c**[**i**]** **=** a**[**i**]\***b**[**i**];**

**}**

This saves all the overhead involved in interpreting the Python code and manipulating Python objects, but at the expense of the benefits gained from coding in Python. Furthermore, the coding work required increases with the dimensionality of our data. In the case of a 2-D array, for example, the C code (abridged as before) expands to

**for** **(**i **=** **0;** i **<** rows**;** i**++)** **{**

**for** **(**j **=** **0;** j **<** columns**;** j**++)** **{**

c**[**i**][**j**]** **=** a**[**i**][**j**]\***b**[**i**][**j**];**

**}**

**}**

NumPy gives us the best of both worlds: element-by-element operations are the “default mode” when an *ndarray* is involved, but the element-by-element operation is speedily executed by pre-compiled C code. In NumPy

c **=** a **\*** b

does what the earlier examples do, at near-C speeds, but with the code simplicity we expect from something based on Python. Indeed, the NumPy idiom is even simpler! This last example illustrates two of NumPy’s features which are the basis of much of its power: vectorization and broadcasting.

Why is NumPy Fast?

Vectorization describes the absence of any explicit looping, indexing, etc., in the code - these things are taking place, of course, just “behind the scenes” in optimized, pre-compiled C code. Vectorized code has many advantages, among which are:

* vectorized code is more concise and easier to read
* fewer lines of code generally means fewer bugs
* the code more closely resembles standard mathematical notation (making it easier, typically, to correctly code mathematical constructs)
* vectorization results in more “Pythonic” code. Without vectorization, our code would be littered with inefficient and difficult to read for loops.

Broadcasting is the term used to describe the implicit element-by-element behavior of operations; generally speaking, in NumPy all operations, not just arithmetic operations, but logical, bit-wise, functional, etc., behave in this implicit element-by-element fashion, i.e., they broadcast. Moreover, in the example above, a and b could be multidimensional arrays of the same shape, or a scalar and an array, or even two arrays of with different shapes, provided that the smaller array is “expandable” to the shape of the larger in such a way that the resulting broadcast is unambiguous. For detailed “rules” of broadcasting see Broadcasting.

Who Else Uses NumPy?

NumPy fully supports an object-oriented approach, starting, once again, with *ndarray*. For example, *ndarray* is a class, possessing numerous methods and attributes. Many of its methods are mirrored by functions in the outer-most NumPy namespace, allowing the programmer to code in whichever paradigm they prefer. This flexibility has allowed the NumPy array dialect and NumPy *ndarray* class to become the *de-facto* language of multi-dimensional data interchange used in Python.

Pandas

pandas is a [Python](https://www.python.org/) package providing fast, flexible, and expressive data structures designed to make working with “relational” or “labeled” data both easy and intuitive. It aims to be the fundamental high-level building block for doing practical, **real-world** data analysis in Python. Additionally, it has the broader goal of becoming **the most powerful and flexible open source data analysis/manipulation tool available in any language**. It is already well on its way toward this goal.

pandas is well suited for many different kinds of data:

* Tabular data with heterogeneously-typed columns, as in an SQL table or Excel spreadsheet
* Ordered and unordered (not necessarily fixed-frequency) time series data.
* Arbitrary matrix data (homogeneously typed or heterogeneous) with row and column labels
* Any other form of observational / statistical data sets. The data need not be labeled at all to be placed into a pandas data structure

The two primary data structures of pandas, [**Series**](https://pandas.pydata.org/docs/reference/api/pandas.Series.html#pandas.Series) (1-dimensional) and [**DataFrame**](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html#pandas.DataFrame) (2-dimensional), handle the vast majority of typical use cases in finance, statistics, social science, and many areas of engineering. For R users, [**DataFrame**](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html#pandas.DataFrame) provides everything that R’s data.frame provides and much more. pandas is built on top of [NumPy](https://numpy.org/) and is intended to integrate well within a scientific computing environment with many other 3rd party libraries.

Here are just a few of the things that pandas does well:

* Easy handling of **missing data** (represented as NaN) in floating point as well as non-floating point data
* Size mutability: columns can be **inserted and deleted** from DataFrame and higher dimensional objects
* Automatic and explicit **data alignment**: objects can be explicitly aligned to a set of labels, or the user can simply ignore the labels and let Series, DataFrame, etc. automatically align the data for you in computations
* Powerful, flexible **group by** functionality to perform split-apply-combine operations on data sets, for both aggregating and transforming data
* Make it **easy to convert** ragged, differently-indexed data in other Python and NumPy data structures into DataFrame objects
* Intelligent label-based **slicing**, **fancy indexing**, and **subsetting** of large data sets
* Intuitive **merging** and **joining** data sets
* Flexible **reshaping** and pivoting of data sets
* **Hierarchical** labeling of axes (possible to have multiple labels per tick)
* Robust IO tools for loading data from **flat files** (CSV and delimited), Excel files, databases, and saving / loading data from the ultrafast **HDF5 format**
* **Time series**-specific functionality: date range generation and frequency conversion, moving window statistics, date shifting, and lagging.

Many of these principles are here to address the shortcomings frequently experienced using other languages / scientific research environments. For data scientists, working with data is typically divided into multiple stages: munging and cleaning data, analyzing / modeling it, then organizing the results of the analysis into a form suitable for plotting or tabular display. pandas is the ideal tool for all of these tasks.

Some other notes

* pandas is **fast**. Many of the low-level algorithmic bits have been extensively tweaked in [Cython](https://cython.org/) code. However, as with anything else generalization usually sacrifices performance. So if you focus on one feature for your application you may be able to create a faster specialized tool.
* pandas is a dependency of [statsmodels](https://www.statsmodels.org/), making it an important part of the statistical computing ecosystem in Python.
* pandas has been used extensively in production in financial applications.

Data structures

|  |  |  |
| --- | --- | --- |
| **Dimensions** | **Name** | **Description** |
| 1 | Series | 1D labeled homogeneously-typed array |
| 2 | DataFrame | General 2D labeled, size-mutable tabular structure with potentially heterogeneously-typed column |

Why more than one data structure?

The best way to think about the pandas data structures is as flexible containers for lower dimensional data. For example, DataFrame is a container for Series, and Series is a container for scalars. We would like to be able to insert and remove objects from these containers in a dictionary-like fashion.

Also, we would like sensible default behaviors for the common API functions which take into account the typical orientation of time series and cross-sectional data sets. When using the N-dimensional array (ndarrays) to store 2- and 3-dimensional data, a burden is placed on the user to consider the orientation of the data set when writing functions; axes are considered more or less equivalent (except when C- or Fortran-contiguousness matters for performance). In pandas, the axes are intended to lend more semantic meaning to the data; i.e., for a particular data set, there is likely to be a “right” way to orient the data. The goal, then, is to reduce the amount of mental effort required to code up data transformations in downstream functions.

For example, with tabular data (DataFrame) it is more semantically helpful to think of the **index** (the rows) and the **columns** rather than axis 0 and axis 1. Iterating through the columns of the DataFrame thus results in more readable code:

**for** col **in** df.columns:

series = df[col]

*# do something with series*

Mutability and copying of data

All pandas data structures are value-mutable (the values they contain can be altered) but not always size-mutable. The length of a Series cannot be changed, but, for example, columns can be inserted into a DataFrame. However, the vast majority of methods produce new objects and leave the input data untouched. In general we like to **favor immutability** where sensible.

Getting support

The first stop for pandas issues and ideas is the [Github Issue Tracker](https://github.com/pandas-dev/pandas/issues). If you have a general question, pandas community experts can answer through [Stack Overflow](https://stackoverflow.com/questions/tagged/pandas).

Community

pandas is actively supported today by a community of like-minded individuals around the world who contribute their valuable time and energy to help make open source pandas possible. Thanks to [all of our contributors](https://github.com/pandas-dev/pandas/graphs/contributors).

If you’re interested in contributing, please visit the [contributing guide](https://pandas.pydata.org/docs/development/contributing.html#contributing).

pandas is a [NumFOCUS](https://numfocus.org/sponsored-projects) sponsored project. This will help ensure the success of the development of pandas as a world-class open-source project and makes it possible to [donate](https://pandas.pydata.org/donate.html) to the project.

Project governance

The governance process that pandas project has used informally since its inception in 2008 is formalized in [Project Governance documents](https://github.com/pandas-dev/pandas-governance). The documents clarify how decisions are made and how the various elements of our community interact, including the relationship between open source collaborative development and work that may be funded by for-profit or non-profit entities.

Wes McKinney is the Benevolent Dictator for Life (BDFL).