

PYTHON MACHINE LEARNING PRO-TIPS

Adam Geitgey

Python Machine Learning Pro Tips

Adam Geitgey

First Edition, Version 1.0.1

Copyright 2018 Adam Geitgey, All Rights Reserved.

This book is written by a self-published author and without the financial support of a publisher. Please don't post it online.

If you received this book and did not purchase it, please consider purchasing your own copy to support the continued existence of self-published authors.

Contents

| | | |
|----------|--|-----------|
| 1 | Why Use Python for Machine Learning? | 6 |
| 2 | Choosing Between Python 2 and Python 3 | 9 |
| 3 | NumPy and Vectorization | 11 |
| | Memory Efficiency | 11 |
| | Automatic Vectorization and Processor Efficiency | 13 |
| | Matrix Operations with NumPy | 16 |
| 4 | Pandas | 17 |
| 5 | Scraping Web Data with Pandas | 23 |
| 6 | Plotting data with Matplotlib | 28 |
| 7 | Speeding Up Data Processing with Parallelism | 33 |
| | The Normal Approach | 34 |
| | Parallelizing Code with Process Pools | 36 |
| | When to Use Process Pools | 40 |
| | Bypassing the Global Interpreter Lock | 41 |
| 8 | Working with Files Using Pathlib | 42 |
| | The Wrong Solution: Building File Paths by Hand | 43 |

Contents

| | |
|---|-----------|
| The Old Solution: Python's os.path Module | 44 |
| The Better Solution: Python 3's Pathlib | 45 |
| 9 Reduce Bugs with Static Type Checking | 48 |
| Preventing Bugs with Type Annotations | 49 |
| How to Declare Types | 51 |
| Declaring Complex Types | 53 |
| Running a Type Checker | 55 |
| When to Use Type Declarations | 56 |

1 Why Use Python for Machine Learning?



Python is the most popular programming language for machine learning and artificial intelligence development and research. Because of this, it's also one of the fastest growing languages overall. It's surpassed even C# and PHP in popularity according to StackOverflow.

But Python is also frequently listed in surveys as one of the languages programmers most want to work in. Never before has a tool so beloved by developers also become the tool most requested by industry. It's a great time to be a Python developer!

I've been writing code in Python since the 1990s. Back then, it was an obscure programming language with a small but vocal fanbase mostly working in the Silicon Valley start-up community. The same design features that made Python great for start-ups and quick prototyping made it a great fit for machine learning.

1 Why Use Python for Machine Learning?

Python was designed from the very beginning to be easy to use. Many developers were first attracted to Python because you could get more done in 5 lines of Python code with 500 lines of C++ or Java code. Python is not only a high-level language with a simple syntax, but it also includes a huge library of built-in functions that let you do almost anything you want right out of the box.

The tradeoff is that programs written in Python aren't terribly fast. Python is an interpreted language that runs slowly compared to a compiled language like C++ or Java. That might make it seem like a bad fit for the heavy computation required for machine learning, but Python had a trick up its sleeves. One of the original design decisions of Python was to make it as easy as possible to expose libraries written in other programming languages inside of Python as if they were Python libraries. This means that all the heavy math can be done inside optimized C code while the Python wrapper code controls the overall flow of the program. This lets us combine the ease of development in Python with the speed of execution of a lower-level programming language.

Python made another decision early on that also significantly contributed to its success in the machine learning field. One of Python's main philosophies is that the "batteries are included." That means that the user should have access to a vast library of functionality just by installing Python itself. Unlike a language like JavaScript that for many years didn't include even basic functionality in the language itself, Python prided itself on making the default installation include everything you needed to write complex programs. Even something as complex as parsing an HTML page to extract data or building a full web server was built into Python's standard library. Once people got a taste for the ease of working with data in Python, they

1 Why Use Python for Machine Learning?

built even more complex add-on libraries like NumPy and scikit-learn that made it easy to build machine learning models in Python with only a tiny amount of code. But because the most complex parts of the code run in C, the performance was still great. Python offered the perfect combination of ease of use and power at a time when a new field of computing was developing. Python and machine learning grew up together and were a natural fit for each other.

Over time, all programming languages will gain their own machine learning features. But right now, there is no better development environment for machine learning than Python. Luckily, Python is also one of the easiest programming languages to learn, so there's no reason not to get started!

2 Choosing Between Python 2 and Python 3

In the history of Python, there is one major controversy that has overshadowed all other controversies. About 10 years ago, the developers behind Python released a major new language revision called Python 3. This version fixed a lot of long-standing design issues with the language itself, especially regarding working with text written in languages other than English. But Python 3 was also the first time that a large number of breaking changes were made to the language syntax all at once.

Because of this, many developers didn't want to upgrade their Python 2 code to run on Python 3. For years, developers kept using Python 2 and actively fighting against moving to Python 3. They thought it was too much work to switch without enough reward.

But the developers behind Python kept adding great new features to Python 3 and all the major software libraries eventually added Python 3 support. It took nearly a decade, but finally, Python 3 has won out. It's a huge win for all of us because Python 3 is a much nicer language—especially if you do natural language processing.

At this point, you should consider Python 2 a dead language. If you don't work for a company that makes you use it, don't waste your time even in-

2 Choosing Between Python 2 and Python 3

stalling it. Download the latest version of Python 3 and use it for all your machine learning work. It's an amazing language with lots of great features. And just be glad that you missed all the drama!

3 NumPy and Vectorization



NumPy is an add-on library for Python. Its main feature is that it lets you create large arrays of data in memory and work with them in a very memory-efficient and processor-efficient way.

When we are writing machine learning code in Python, we'll almost always use NumPy arrays instead of Python's native arrays to load and process training data.

Memory Efficiency

Python is a very high-level programming language. It's designed to be easy for humans to work with at the expense of running more slowly and using more memory than other programming languages. Python's version of arrays, called lists, are a good example of this.

3 NumPy and Vectorization

Python lists are very easy to work with. You can create them and resize them whenever you want and you can add and remove items from them without any restrictions.

```
1 # Create a list with 3 numbers in it
2 l = [3, 1, 2]
3
4 # Add another item to the list
5 l.append(4)
6
7 # Print out the sum of the numbers in the list
8 print(sum(l))
9
10 # The output is "10"
```

But the way that Python lists are stored in memory isn't very efficient. When you have multi-dimensional arrays with millions of elements (like you often have with training data), the overhead of working with Python lists becomes a real problem.

NumPy is designed specifically to work with large arrays of data. It allocates blocks of contiguous memory and it can quickly navigate through the entire array much more quickly than with a Python list.

To use NumPy, you first have to import it. Then, instead of creating a Python list, you declare your arrays as NumPy arrays. From there, you can do most of the same things you can do with a Python list, but it requires using NumPy's helper functions.

3 NumPy and Vectorization

```
1 import numpy as np
2
3 # Create a numpy array with 4 numbers in it.
4 a = np.array([3, 1, 2, 4])
5
6 # Print the sum of the numbers in the list.
7 print(np.sum(a))
8
9 # The output is "10"
```

When you are working with a small amount of data, the extra hassle of learning the slightly different NumPy syntax probably isn't worth it. But when you are working with an array of thousands or millions of items, you will save a huge amount of memory by using NumPy instead of normal Python Lists.

Automatic Vectorization and Processor Efficiency

There's another big performance reason to use NumPy—it can also perform mathematical operations on arrays of data much, much faster than Python itself can. That might seem impossible. After all, Python runs on the same CPU as NumPy. How can NumPy multiply a number faster than Python?

The secret is vectorization. Modern CPUs are capable of doing multiple operations in parallel. If you tell the CPU that you need to multiply lots of numbers and give it all the numbers to multiply, the CPU can multiply them simultaneously in large batches. This is called *vectorization* because the CPU can perform mathematical operations on entire vectors, or lists,

3 NumPy and Vectorization

of numbers in one pass.

Most normal programming languages either don't support this at all or they attempt to silently optimize loops to vectorize the code for you automatically. But for machine learning, needing to multiply lots of numbers at once is so common that it's worth having the ability to control this yourself.

Let's say we have a list of five numbers and we want to multiply all of them by 0.5. The normal way to do that is to use a **for** loop.

```
1 numbers = [  
2     10.5,  
3     5.8,  
4     34.1,  
5     17.4,  
6     32.1  
7 ]  
8  
9 # Loop over each number  
10 for i, value in enumerate(numbers):  
11     # Multiply the current number by 0.5  
12     # and save the result  
13     numbers[i] = value * 0.5  
14  
15 # Print the final array with the multiplied values  
16 print(numbers)
```

But that's an incredibly inefficient way to use your CPU. All CPUs made in the last few years have special extensions that let them multiply batches of numbers in one pass. But since we are giving the CPU one number at a time, it can't optimize this operation.

Here's how the same code looks in NumPy:

3 NumPy and Vectorization

```
1 import numpy as np
2
3 # Create a numpy array
4 numbers = np.array([
5     10.5,
6     5.8,
7     34.1,
8     17.4,
9     32.1
10 ])
11
12 # Multiply all array elements separately by 0.5
13 numbers = numbers * 0.5
14
15 # Print the final array with the multiplied values
16 print(numbers)
```

The final result is exactly the same. But NumPy is smart enough to know that since you are multiplying an entire array by a fixed value, that it should parallelize that operation and take advantage of any CPU optimizations that might speed that up.

When writing machine learning code, it's almost always a mistake to write a **for** loop that loops over each element of an array. There's usually a much more efficient way to write the same operation using a library like NumPy. This process of taking a **for** loop and figuring out how to express it in NumPy without using any loops is called **vectorizing your code** and it's a key skill that you'll develop with practice as you become more familiar with NumPy.

Matrix Operations with NumPy

Not only are NumPy arrays much more efficient than Python arrays, but they also can do a lot of things that normal Python lists can't do.

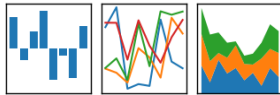
NumPy arrays support common matrix operations from linear algebra. For example, you can easily calculate dot products with `np.dot()`. And with the `np.linalg` package, you can do more advanced calculations like finding determinants or calculating eigenvalues. And not only are these operations easy to do, but they are implemented in an efficient way.

Linear algebra is the foundation of many machine learning algorithms. Because NumPy makes this stuff so easy, almost all machine learning libraries on Python use NumPy arrays as the standard format for passing in data in returning results. It's the basic foundation for nearly all other Python machine learning libraries.

4 Pandas

pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



When you work on machine learning projects, you'll spend a lot of your time exploring and cleaning up tabular data. Pandas is a Python library that makes this a lot easier.

Let's say that we have a spreadsheet with population data.

| City | Population | Country |
|-----------|------------|---------|
| Chongqing | 30,165,500 | China |
| Shanghai | 24,183,300 | China |
| Beijing | 21,707,000 | China |
| Lagos | 16,060,303 | Nigeria |
| Istanbul | 15,029,231 | Turkey |

The data is a mix of numbers and text. It's easy to work with in a spreadsheet but it can be difficult to manipulate in a program. Pandas makes it simple to work with data like this in a program.

The main feature of pandas is a class called a `DataFrame`. A `DataFrame` is an object that acts like an invisible spreadsheet. You can load data into

4 Pandas

it from a file and then do all the normal kinds of operations you do inside a spreadsheet like sorting the data, filtering it, and even building pivot tables.

The first step is to import pandas and load the data into a `DataFrame`.

```
1 # Import the pandas library.
2 import pandas as pd
3
4 # Load the data from a CSV file into a DataFrame.
5 # You can also load Excel files and other common formats.
6 df = pd.read_csv("population_data.csv")
7
8 # Print out what we loaded.
9 print(df)
```

Running this will show the data we loaded.

```
1           City  Population  Country
2  0    Chongqing    30165500    China
3  1    Shanghai    24183300    China
4  2     Beijing    21707000    China
5  3      Lagos    16060303  Nigeria
6  4    Istanbul    15029231    Turkey
7
8  ... etc ...
```

Now we can access any column of data by name. Pandas automatically knows the names of each column based on the header row in the CSV file.

```
1 # Print out just the "City" column
2 print(df['City'])
```

That will print out just the cities.

4 Pandas

```
1 [87 rows x 3 columns]
2 0          Chongqing
3 1          Shanghai
4 2          Beijing
5 3          Lagos
6 4          Istanbul
7
8 ... etc ...
```

We can also use the `df.describe()` function to get an overview of our data.

```
1 # Get basic statistics for the whole DataFrame
2 print(df.describe())
```

This produces a list of basic population statistics automatically.

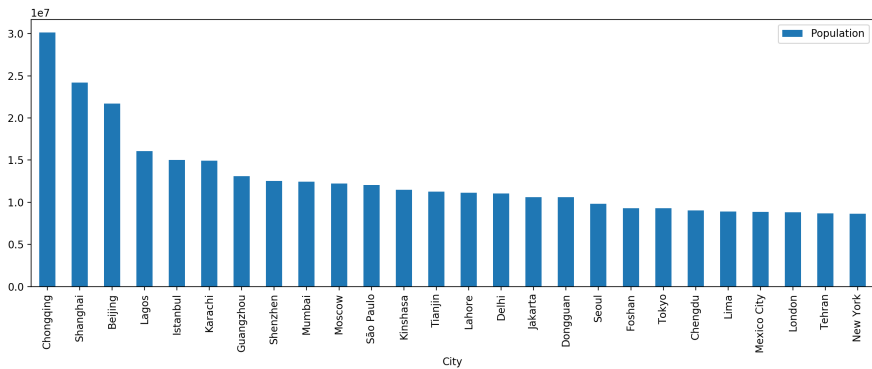
```
1          Population
2 count      87.00000
3 mean    7421527.4483
4 std     4726676.1679
5 min     3002645.0000
6 25%     3847477.5000
7 50%     6694000.0000
8 75%     8884500.0000
9 max     30165500.0000
```

Pretty handy! We can even display this data on an interactive graph.

4 Pandas

```
1 # Plot the data on a bar graph.
2 # Use the 'City' column as labels for each bar.
3 df.plot(x="City", kind="bar")
4
5 # Ask pyplot to show the graph on the screen
6 plt.show()
```

This code will pop open a window with a nice graph.



And of course, pandas makes it simple to filter, sort, or process our data further. Let's say we wanted to find out which country had the most cities on the list. That's as easy as grouping the data by country and counting.

```
1 # Count how many times each country appears in the list
2 print(df.groupby("Country").count())
```

That will tell us that China has nine countries on our list.

We could also grab a list of countries that appear at least once by checking for unique values.

4 Pandas

```
1 # Get a list of unique countries on the list
2 print(df["Country"].unique())
```

That gives us all the unique countries.

```
1 [
2   'China' 'Nigeria' 'Turkey'
3   'Pakistan' 'India' 'Russia'
4   'Brazil'
5   'Democratic Republic of the Congo'
6   'Indonesia' 'South Korea' 'Japan'
7   'Peru' 'Mexico' 'United Kingdom'
8   'Iran' 'United States'
9 ]
```

Pandas can also write out data for you. Let's say we wanted to turn this population data into JSON objects that are easy to parse in a javascript program. We can do that with the `to_json()` function.

```
1 # Save the data to a json file
2 df.to_json("pop_data.json", orient="records")
```

That will create a structured JSON file that looks like this:

4 Pandas

```
1  [  
2    {  
3      "City": "Chongqing",  
4      "Population": 30165500,  
5      "Country": "China"  
6    },  
7    {  
8      "City": "Shanghai",  
9      "Population": 24183300,  
10     "Country": "China"  
11   }  
12 ]
```

This is just the beginning of what you can do with pandas. Spending some time reading the pandas documentation and exploring its features will pay off handsomely in the long run!

5 Scraping Web Data with Pandas

Let’s say we are searching the web for a list of the capitals of every country and we stumble across a web page like this:

List of national capitals

From Wikipedia, the free encyclopedia

This is a **list of national capitals**, including capitals of [territories](#) and [dependencies](#), non-sovereign states including [associated states](#) and entities whose [sovereignty](#) is [disputed](#). Sovereign states and observer states within the [United Nations](#) are shown in **bolded** text.

| City | Country | Notes |
|-------------|---|---|
| Abu Dhabi |  United Arab Emirates | |
| Abuja |  Nigeria | Lagos was the capital from 1914 to 1991 |
| Accra |  Ghana | |
| Adamstown |  Pitcairn | British Overseas Territory |
| Addis Ababa |  Ethiopia | |
| Algiers |  Algeria | |
| Alofi |  Niue | Self-governing in free association with New Zealand |

Figure 5.1: Wikipedia’s list of National Capitals

It’s exactly what we want—an up-to-date page with exactly the data that we need!

But the bad news is that the table we want lives inside a web page and there’s no API that we can use to grab just that raw data. So now we have to waste 30 minutes throwing together a crappy script to scrape the data. It’s not hard, but it’s a waste of time that we could spend on something more useful—and somehow 30 minutes always ends up being two hours.

5 Scraping Web Data with Pandas

For me, this kind of thing happens all the time. Luckily, there's a super simple answer. Pandas has a built-in method to scrape tabular data from HTML pages called `read_html()`.

```
1 import pandas as pd
2
3 # Read an HTML page and extract any tables
4 # of data as DataFrames
5 tables = pd.read_html("https://en.wikipedia.org/wiki/
    List_of_national_capitals")
6
7 # Grab the second table on the page (the one we want)
8 country_table = tables[1]
9
10 # Print it
11 print(country_table)
```

It's that simple! Pandas will find any significant HTML tables on the page and return each one as a new DataFrame object. We just need to grab the one that we want.

To upgrade our program from *toy* to *real*, let's tell pandas that row 0 of the table has column headers. We'll also only print out the "City" and "Country" columns to make the output easier to read.

5 Scraping Web Data with Pandas

```
1 import pandas as pd
2
3 # Read an HTML page and extract any tables of data as
  DataFrames.
4 # Assume the first row of each table has the column
  labels.
5 tables = pd.read_html("https://en.wikipedia.org/wiki/
  List_of_national_capitals", header=0)
6
7 # Grab the second table on the page (the one we want)
8 country_table = tables[1]
9
10 # Print just the "City" and "Country" columns
11 print(country_table[["City", "Country"]])
```

Running this gives us this beautiful output:

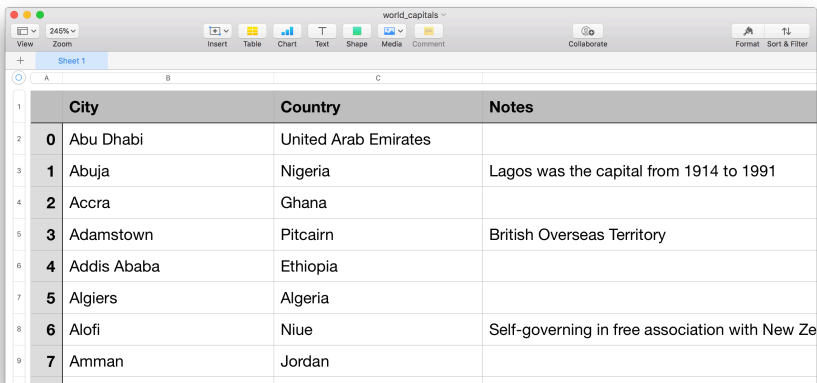
| | City | Country |
|---|-------------|----------------------|
| 0 | Abu Dhabi | United Arab Emirates |
| 1 | Abuja | Nigeria |
| 2 | Accra | Ghana |
| 3 | Adamstown | Pitcairn |
| 4 | Addis Ababa | Ethiopia |
| 7 | | |
| 8 | .. etc .. | |

And since the data now lives in a DataFrame, the world is yours. We can save it out to a CSV or Excel file with one more line of code.

5 Scraping Web Data with Pandas

```
1 import pandas as pd
2
3 df = pd.read_html("https://en.wikipedia.org/wiki/
    List_of_national_capitals", header=0)[1]
4
5 print(df.to_csv("cities.csv"))
```

Here's what that CSV file looks like when I open it up in my spreadsheet application:



The screenshot shows a spreadsheet application window titled "world_capitals". The spreadsheet has three columns: "City", "Country", and "Notes". The data is as follows:

| | City | Country | Notes |
|---|-------------|----------------------|---|
| 0 | Abu Dhabi | United Arab Emirates | |
| 1 | Abuja | Nigeria | Lagos was the capital from 1914 to 1991 |
| 2 | Accra | Ghana | |
| 3 | Adamstown | Pitcairn | British Overseas Territory |
| 4 | Addis Ababa | Ethiopia | |
| 5 | Algiers | Algeria | |
| 6 | Alofi | Niue | Self-governing in free association with New Zealand |
| 7 | Amman | Jordan | |

We can even convert the data to structured JSON.

```
1 import pandas as pd
2
3 df = pd.read_html("https://en.wikipedia.org/wiki/
    List_of_national_capitals", header=0)[1]
4
5 print(df.to_json(orient="records"))
```

If you run that, you'll get this beautiful JSON output:

5 Scraping Web Data with Pandas

```
1  [  
2    {  
3      "City": "Abu Dhabi",  
4      "Country": "United Arab Emirates",  
5      "Notes": null  
6    },  
7    {  
8      "City": "Abuja",  
9      "Country": "Nigeria",  
10     "Notes": "Lagos was the capital from 1914 to 1991"  
11   }  
12 ]
```

So the next time you are looking for raw data on the web, try using pandas to grab it!

6 Plotting data with Matplotlib



Being able to visualize data easily is one of the most useful tools we have at our disposal. Matplotlib allows us to view an interactive graph or chart of almost anything with just a couple of lines of code.

Let's say we have sales data that looks like this:

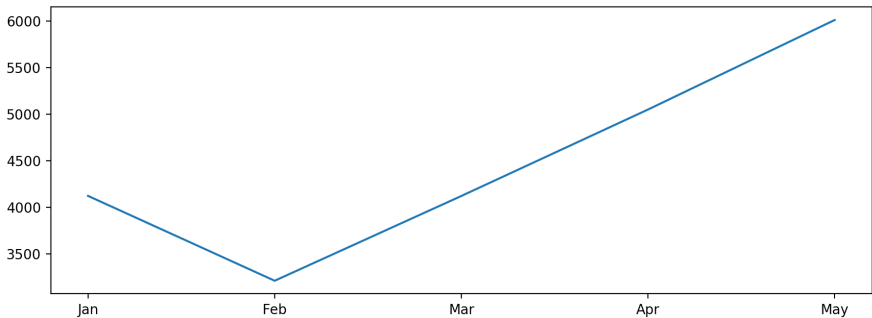
```
1 months = ["Jan", "Feb", "Mar", "Apr", "May"]
2 sales_by_month = [4123, 3212, 4122, 5050, 6011]
```

To visualize that, we just need to import `matplotlib` and call the `plot()` function.

6 Plotting data with Matplotlib

```
1 # Importing matplotlib as plt is the normal convention.
2 import matplotlib.pyplot as plt
3
4 # Sales data
5 months = ["Jan", "Feb", "Mar", "Apr", "May"]
6 sales_by_month = [4123, 3212, 4122, 5050, 6011]
7
8 # Plot the sales data!
9 plt.plot(months, sales_by_month)
10
11 # Display the chart on the screen
12 plt.show()
```

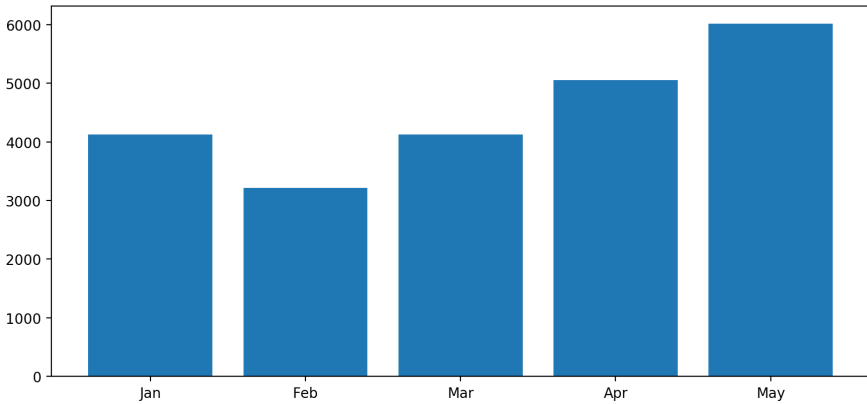
That will open a chart that looks like this:



But for monthly data, a bar graph probably makes more sense. We can do that by just calling `plt.bar()` instead.

```
1 # Plot the sales data as a bar graph
2 plt.bar(months, sales_by_month)
3
4 # Display the chart on the screen
5 plt.show()
```

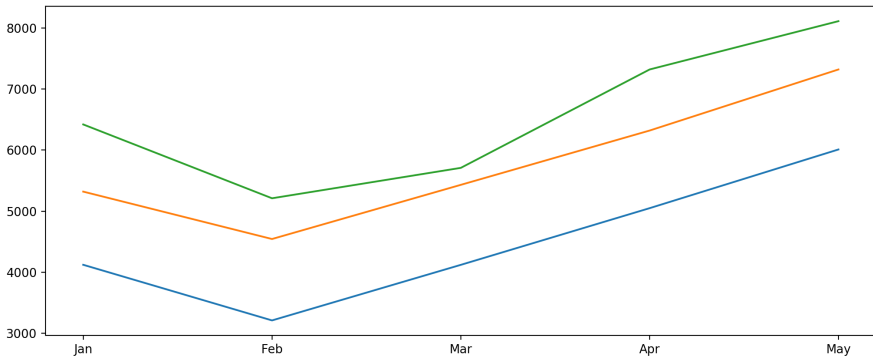
6 Plotting data with Matplotlib



Matplotlib is very flexible. Let's say we want to go a step further and plot year-over-year sales on the same graph. We can do that just by calling `plt.plot()` multiple times—once for each year of data.

```
1 # Sales data
2 months = ["Jan", "Feb", "Mar", "Apr", "May"]
3 sales_2018 = [4123, 3212, 4122, 5050, 6011]
4 sales_2019 = [5321, 4545, 5432, 6321, 7321]
5 sales_2020 = [6421, 5212, 5710, 7321, 8112]
6
7 # Plot the sales data!
8 plt.plot(months, sales_2018)
9 plt.plot(months, sales_2019)
10 plt.plot(months, sales_2020)
11
12 # Display the chart on the screen
13 plt.show()
```

6 Plotting data with Matplotlib



Sometimes it's helpful to save charts to a file so that you can share them with other people. That's as easy as calling `plt.savefig()` and passing in a filename.

```
1 # Import Matplotlib. Calling it plt is the normal
  convention.
2 import matplotlib.pyplot as plt
3
4 # Sales data
5 months = ["Jan", "Feb", "Mar", "Apr", "May"]
6 sales_by_month = [4123, 3212, 4122, 5050, 6011]
7
8 # Plot the sales data!
9 plt.plot(months, sales_by_month)
10
11 # Save the chart to an image file
12 plt.savefig("sales_chart.png")
```

With some practice, you can become a Matplotlib wizard who can spit out 3D charts with ease. But most of the time, just being able to bring up a simple line graph is all you need. So make sure you get familiar with the basic Matplotlib syntax and feel comfortable using it. It's an essential tool

6 *Plotting data with Matplotlib*

in your toolbelt.

7 Speeding Up Data Processing with Parallelism

Python is a great programming language for crunching data and automating repetitive tasks. Got a few gigs of web server logs to process or a million images that need resizing? No problem! You can almost always find a helpful Python library that makes the job easy.

But Python is not always the quickest to run. By default, Python programs execute as a single process using a single CPU. If you have a computer made in the last decade, there's a good chance it has 4 *or more* CPU cores. That means that 75 percent or more of your computer's power is sitting there nearly idle while you are waiting for your program to finish running.

To take advantage of that idle computing power, we need to write our programs in a way where they can do multiple tasks in parallel. In the old days, that was very difficult. But thanks to Python's `concurrent.futures` module, you can parallelize many kinds of data processing jobs with just a couple of lines of code.

The Normal Approach

Let's say we have a folder full of photos and we want to create thumbnails of each photo.

Here's a short program that uses Python's built-in glob function to get a list of all the jpeg files in a folder and then uses the Pillow image processing library to save out 128-pixel thumbnails of each photo.

```
1  import glob
2  import os
3  from PIL import Image
4
5  def make_image_thumbnail(filename):
6      # The thumbnail will be named
7      # <original_filename>_thumbnail.jpg
8      base_filename, file_extension = os.path.splitext(
9          filename
10     )
11     thumbnail_filename = f"{base_filename}_thumbnail{
12         file_extension}"
13
14     # Create and save thumbnail image
15     image = Image.open(filename)
16     image.thumbnail(size=(128, 128))
17     image.save(thumbnail_filename, "JPEG")
18     return thumbnail_filename
19
20 # Loop through all jpeg files in the
21 # folder and make a thumbnail for each
22 for image_file in glob.glob("*.jpg"):
23     thumbnail_file = make_image_thumbnail(image_file)
24     print(f"{image_file} saved as {thumbnail_file}")
```

7 Speeding Up Data Processing with Parallelism

This program follows a pattern that we'll see often in data processing scripts:

1. We start with a list of files (or other data) that we want to process.
2. We write a helper function that can process one piece of that data.
3. We process each piece of data, one at a time, using a **for** loop to call the helper function.

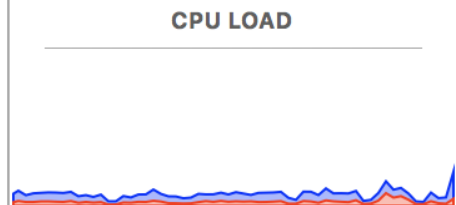
Let's test this program on a folder with 1,000 jpeg files and see how long it takes to run.

```
1 $ time python3 thumbnails_1.py
2
3 1430028941_4db9dedd10.jpg saved as 1430028941
   _4db9dedd10_thumbnail.jpg
4
5 [... about 1000 more lines of output ...]
6
7 user 0m7.086s
8 sys 0m0.743s
```

The program took 8.9 seconds to run. But how hard was the computer working?

Let's run the program again and check Activity Monitor while it's running.

| | |
|---------|--------|
| System: | 5.62% |
| User: | 18.50% |
| Idle: | 75.88% |



The computer is 75 percent idle! What's up with that?

7 *Speeding Up Data Processing with Parallelism*

The problem is that my computer has four CPU cores, but Python is only using one of them. So while I'm maxing out the capacity of one CPU, the other three CPUs aren't doing anything. I need a way to split the workload into four separate chunks that I can run in parallel.

Parallelizing Code with Process Pools

Here's a simple approach we can use to process our data in parallel:

1. Split the list of jpeg files into four equal chunks.
2. Run four separate instances of the Python interpreter.
3. Have each instance of Python process one of the four chunks of data.
4. Combine the results from the four processes to get the final list of results.

Four copies of Python running on four separate CPU cores should be able to do roughly four times as much work as one CPU, right?

The neat part is that Python handles all the setup work for us. We just tell it which function we want to run and how many instances of Python to use, and it does the rest. We only have to change three lines of code to get this working.

First, we need to import the `concurrent.futures` library.

```
1 import concurrent.futures
```

Next, we need to tell Python to boot up four extra Python instances. We do that by telling it to create a `ProcessPool`.

7 Speeding Up Data Processing with Parallelism

```
1 with concurrent.futures.ProcessPoolExecutor() as  
  executor:
```

By default, it will create one Python process for each CPU in your machine. So if you have four CPUs, this will start up four Python processes.

The final step is to ask the Process Pool to execute our helper function on our list of data using those four processes. We can do that by replacing the original **for** loop we had.

```
1 for image_file in glob.glob("*.jpg"):  
2     thumbnail_file = make_image_thumbnail(image_file)
```

With this new call to `executor.map()`:

```
1 image_files = glob.glob("*.jpg")  
2 for image_file, thumbnail_file in zip(  
3     image_files, executor.map(  
4         make_image_thumbnail,  
5         image_files  
6     )  
7 )
```

The `executor.map()` function takes in the helper function to call and the list of data to process with it. It does all the hard work of splitting up the list, sending the sub-lists off to each child process, running the child processes, and combining the results.

The `executor.map()` function returns a list of results from each time our helper function was called. It returns the results in the same order as the list of data we gave it to process. So I've used Python's `zip()` function as a shortcut to grab the original filename and the matching result in one step.

7 Speeding Up Data Processing with Parallelism

Here's how our program looks with those three changes:

7 Speeding Up Data Processing with Parallelism

```
1 import glob
2 import os
3 from PIL import Image
4 import concurrent.futures
5
6 def make_image_thumbnail(filename):
7     # The thumbnail will be named
8     # <original_filename>_thumbnail.jpg
9     base_filename, file_extension = os.path.splitext(
10         filename)
11     thumbnail_filename = f"{base_filename}_thumbnail{
12         file_extension}"
13
14     # Create and save thumbnail image
15     image = Image.open(filename)
16     image.thumbnail(size=(128, 128))
17     image.save(thumbnail_filename, "JPEG")
18     return thumbnail_filename
19
20 # Create a pool of processes. By default, one
21 # is created for each CPU in your machine.
22 with concurrent.futures.ProcessPoolExecutor() as executor
23 :
24     # Get a list of files to process
25     image_files = glob.glob("*.jpg")
26
27     # Process the list of files, but split
28     # the work across the process pool to use all CPUs!
29     for image_file, thumbnail_file in zip(
30         image_files,
31         executor.map(
32             make_image_thumbnail,
33             image_files
34         )
35     ):
36         print(f"{image_file} saved as {thumbnail_file}")
```

7 Speeding Up Data Processing with Parallelism

Let's run the program again and see if it finishes more quickly this time.

```
1 $ time python3 thumbnails_2.py
2 1430028941_4db9dedd10.jpg saved as 1430028941
   _4db9dedd10_thumbnail.jpg
3
4 [... about 1000 more lines of output ...]
5
6 real 0m2.274s
7 user 0m8.959s
8 sys 0m0.951s
```

It finished in 2.2 seconds! That's a 4x speedup over the original version. The elapsed time was faster because we are using four CPUs instead of just one.

But if you look closely, you'll see that the “user” time was almost nine seconds. How did the program finish in 2.2 seconds but still somehow run for nine seconds?

That's because the user time is a sum of CPU time across *all CPUs*. We completed the same nine seconds of work as last time—but we finished it with four CPUs in only 2.2 real-world seconds.

When to Use Process Pools

Using process pools is a great solution when we have a list of data to process and each piece of data can be processed independently. Here are some examples:

- Grabbing statistics out of a collection of separate web server log files
- Parsing data out of a bunch of XML, CSV, or JSON files

7 *Speeding Up Data Processing with Parallelism*

- Pre-processing lots of images to create a machine learning data set

But process pools aren't always the answer. Using a process pool requires passing data back and forth between Python processes. If the data we are working with can't be efficiently passed between processes, this won't work. For example, this isn't a great fit for working with streaming video. Our computer will spend more time passing each relatively large video frame to a worker process than it will save by splitting up the work of processing each frame between CPUs.

Also, the data won't be processed in a predictable order since multiple CPUs will be processing different chunks of our data at the same time. If we need the result from processing the previous piece of data to process the next piece of data, this approach won't work.

Bypassing the Global Interpreter Lock

You might have heard that Python has a **global interpreter lock**, or **GIL**. That means that even if your program is multi-threaded, only one instruction of Python code can be executed at once by any thread. In other words, multi-threaded Python code can't truly run in parallel.

But process pools work around this issue! Because we are running truly separate Python instances, each instance has its own GIL. You get true parallel execution of Python code at the cost of some extra overhead.

With the `concurrent.futures` library, Python gives you a simple way to tweak your scripts to use all the CPU cores in your computer at once. Don't be afraid to try it out. Once you get the hang of it, it's as simple as using a `for` loop, but it can cut your program's runtime down massively.

8 Working with Files Using Pathlib

One of programming's little annoyances is that Microsoft Windows uses a backslash character between folder names while almost every other computer uses a forward slash.

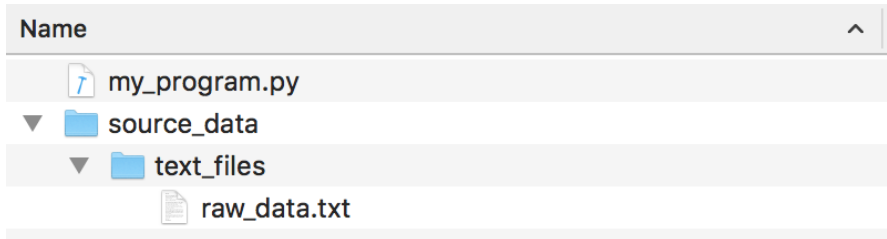
```
1 Windows filenames:
2 C:\some_folder\some_file.txt
3
4 Most other operating systems:
5 /some_folder/some_file.txt
```

This is an accident of early 1980s computer history. The first version of MS-DOS used the forward slash character for specifying command-line options. When Microsoft added support for folders in MS-DOS 2.0, the forward slash character was already taken so they used a backslash instead. Several decades later, we are still stuck with this incompatibility.

If you want your machine learning code to work on both Windows and Mac/Linux, you'll need to deal with these kinds of platform-specific issues. Luckily, Python 3 has a module called **pathlib** that makes working with file paths nearly painless.

The Wrong Solution: Building File Paths by Hand

Let's say you have a data folder that contains a file that you want to open in your Python program.



This is the *wrong* way to code it in Python:

```
1 data_folder = "source_data/text_files/"
2
3 file_to_open = data_folder + "raw_data.txt"
4
5 f = open(file_to_open)
6
7 print(f.read())
```

Technically this code will still work on Windows because Python has a hack where it will recognize either kind of slash when you call **open()** on Windows. But even still, you shouldn't depend on that. Not all Python libraries will work if you use the wrong kind of slash on the wrong operating system—especially if they interface with external programs or libraries.

And Python's support for mixing slash types is a Windows-only hack that doesn't work in reverse. Using backslashes in code will fail on a Mac:

8 Working with Files Using Pathlib

```
1 data_folder = r"source_data\text_files\  
2  
3 file_to_open = data_folder + "raw_data.txt"  
4  
5 f = open(file_to_open)  
6  
7 print(f.read())
```

On a Mac, this code will throw a `FileNotFoundError` exception.

For all these reasons and more, writing code with hard-coded path strings is the kind of thing that will make other programmers look at you with great suspicion. You should avoid it.

The Old Solution: Python's `os.path` Module

Python's **`os.path`** module has lots of tools for working around these kinds of operating system-specific file system issues.

You can use **`os.path.join()`** to build a path string using the right kind of slash for the current operating system.

```
1 import os.path  
2  
3 data_folder = os.path.join("source_data", "text_files")  
4  
5 file_to_open = os.path.join(data_folder, "raw_data.txt")  
6  
7 f = open(file_to_open)  
8  
9 print(f.read())
```

This code will work perfectly on both Windows or Mac. The problem is that it's a pain to use. Writing out `os.path.join()` and passing in each part of the path as a separate string is wordy and unintuitive.

Since most of the functions in the `os.path` module are similarly annoying to use, developers often *forget* to use them even when they know better. This leads to a lot of cross-platform bugs and angry users.

The Better Solution: Python 3's Pathlib

Python 3.4 introduced a new standard library for dealing with files and paths called **pathlib**! To use it, you just pass a path or filename into a `Path()` object using forward slashes and it handles the rest.

```
1 from pathlib import Path
2
3 data_folder = Path("source_data/text_files/")
4
5 file_to_open = data_folder / "raw_data.txt"
```

Notice two things here:

1. You should use forward slashes with **pathlib** functions. The `Path()` object will convert forward slashes into the correct kind of slash for the current operating system. Nice!
2. If you want to add on to the path, you can use the `/` operator directly in your code. No need to type out `os.path.join(a, b)` over and over.

But beyond making it easier to construct paths, Pathlib also provides helpful utility functions that we'll use over and over. For example, we can use

8 Working with Files Using Pathlib

it to read this text file with one more line of code.

```
1 print(file_to_open.read_text())
```

In fact, `pathlib` makes most standard file operations quick and easy.

```
1 from pathlib import Path
2
3 filename = Path("source_data/text_files/raw_data.txt")
4
5 print(filename.name)
6 # prints "raw_data.txt"
7
8 print(filename.suffix)
9 # prints ".txt"
10
11 print(filename.stem)
12 # prints "raw_data"
13
14 if not filename.exists():
15     print("Oops, file doesn't exist!")
16 else:
17     print("Yay, the file exists!")
```

You can even use `pathlib` to explicitly convert a Unix path into a Windows-formatted path.

8 Working with Files Using Pathlib

```
1 from pathlib import Path, PureWindowsPath
2
3 filename = Path("source_data/text_files/raw_data.txt")
4
5 # Convert path to Windows format
6 path_on_windows = PureWindowsPath(filename)
7
8 print(path_on_windows)
9 # prints "source_data\text_files\raw_data.txt"
```

If you want to get fancy, you can even use `pathlib` to do things like resolve relative file paths, parse network share paths, and generate `file://` URLs. Here's an example that will open a local file in your web browser with just two lines of code.

```
1 from pathlib import Path
2 import webbrowser
3
4 filename = Path("source_data/text_files/raw_data.txt")
5
6 webbrowser.open(filename.absolute().as_uri())
```

Anytime you need to work with files on disk, see if `pathlib` is a fit. It's almost always the easiest option.

9 Reduce Bugs with Static Type Checking

One of the most common complaints about the Python programming language is that variables are dynamically typed. That means you declare variables without giving them a specific data type. Types are automatically assigned at runtime based on what data is passed in.

```
1 president_name = "Franklin Delano Roosevelt"
2 print(type(president_name))
3
4 # Result is:
5 # <class 'str'>
```

In this case, the variable `president_name` is created as `str` type because we passed in a string. But Python didn't know it would be a string until it actually ran that line of code.

By comparison, a language like Java is statically typed. To create the same variable in Java, you have to declare the string explicitly with a `String` type.

```
1 String president_name = "Franklin Delano Roosevelt";
```

Because Java knows ahead of time that `president_name` can only hold a

9 Reduce Bugs with Static Type Checking

`String`, it will give you a compile error if you try to do something silly like store an integer in it or pass it into a function that expects something other than a `String`.

Preventing Bugs with Type Annotations

It's usually faster to write new code in a dynamically typed language like Python because you don't have to write out the type declarations by hand. But when your codebase gets large, you'll inevitably run into runtime bugs that static typing would have prevented.

Here's an example of a common type of bug in Python:

```
1 def get_first_name(full_name):
2     return full_name.split(" ")[0]
3
4 fallback_name = {
5     "first_name": "UserFirstName",
6     "last_name": "UserLastName"
7 }
8
9 raw_name = input("Please enter your name: ")
10 first_name = get_first_name(raw_name)
11
12 # If the user didn't type anything in, use
13 # the fallback name
14 if not first_name:
15     first_name = get_first_name(fallback_name)
16
17 print(f"Hi, {first_name}!")
```

All we are doing is asking the user for their name and then printing out

9 Reduce Bugs with Static Type Checking

Hi, < first name >!. And if the user doesn't type anything, we want to print out Hi, UserFirstName! as a fallback.

This program will work perfectly if you run it and type in a name, but it will crash if you leave the name blank.

```
1 Traceback (most recent call last):
2   File "test.py", line 14, in <module>
3     first_name = get_first_name(fallback_name)
4   File "test.py", line 2, in get_first_name
5     return full_name.split(" ")[0]
6 AttributeError: 'dict' object has no attribute 'split'
```

The problem is that `fallback_name` isn't a string—it's a `dict` object. So the `get_first_name` function fails because we passed in an object that doesn't have a `.split()` function.

It's a simple and obvious bug to fix, but what makes this bug *insidious* is that you will never know that the bug exists until a user happens to run the program and leave the name blank. You might test the program a thousand times yourself and never notice this simple bug because you always typed in a name.

Static typing prevents this kind of bug. Before you even try to run the program, static typing will tell you that you can't pass `fallback_name` into `get_first_name()` because it expects a **str** but you are giving it a **dict**. Your code editor can even highlight the error as you type!

When this kind of bug happens in Python, it's usually not in a simple function like this. The bug is usually buried several layers down in the code and triggered because the data passed in is slightly different than previously expected. To debug it, you have to recreate the user's input and figure out

9 Reduce Bugs with Static Type Checking

where it went wrong. *So much time* is wasted debugging these easily preventable bugs.

The good news is that you can now use static typing in Python—and it's an opt-in feature.

How to Declare Types

Let's update the buggy program by declaring the type of each variable and each function input/output.

Here's the updated version:

```
1  from typing import Dict
2
3  def get_first_name(full_name: str) -> str:
4      return full_name.split(" ")[0]
5
6  fallback_name: Dict[str, str] = {
7      "first_name": "UserFirstName",
8      "last_name": "UserLastName"
9  }
10
11 raw_name: str = input("Please enter your name: ")
12 first_name: str = get_first_name(raw_name)
13
14 # If the user didn't type anything in, use
15 # the fallback name
16 if not first_name:
17     first_name = get_first_name(fallback_name)
18
19 print(f"Hi, {first_name}!")
```

9 Reduce Bugs with Static Type Checking

In Python 3.6, you declare a variable type like this:

```
1 variable_name: type
```

If you are assigning an initial value when you create the variable, it's as simple as this:

```
1 my_string: str = "My String Value"
```

And you declare a function's input and output types like this:

```
1 def function_name(parameter1: type) -> return_type:
```

It's pretty simple—just a small tweak to the normal Python syntax. But now that the types are declared, look what happens when I run the type checker:

```
1 $ mypy typing_test.py
2
3 test.py:16: error: Argument 1 to "get_first_name" has
   incompatible type Dict[str, str]; expected "str"
```

Without even executing the program, it knows there's no way that line 16 will work! You can fix the error right now without waiting for a user to discover it three months from now.

And if you are using an IDE like PyCharm, it will automatically check types and show you where something is wrong before you even hit “Run.”

9 Reduce Bugs with Static Type Checking

```
11 raw_name: str = input("Please enter your name: ")
12 first_name: str = get_first_name(raw_name)
13
14 # If the user didn't type anything in, use the fallback name
15 if not first_name:
16     first_name = get_first_name(fallback_name)
17
18 print(f"Hi, {first_name}")
```

Expected type 'str', got 'Dict[str, str]' instead [more...](#) (%F1)

Declaring Complex Types

Declaring types for `str` or `int` variables is simple. The headaches happen when you are working with more complex data types like nested lists and dictionaries. Luckily Python 3.6's syntax for this isn't too bad—at least not for a language that added typing as an afterthought.

The basic pattern is to import the name of the complex data type from the `typing` module and then pass in the nested types in brackets.

`Dict`, `List`, and `Tuple` are the most common complex data types you'll use. Here's what it looks like to use them:

9 Reduce Bugs with Static Type Checking

```
1 from typing import Dict, List
2
3 # A dictionary where the keys are strings
4 # and the values are ints
5
6 name_counts: Dict[str, int] = {
7     "Adam": 10,
8     "Guido": 12
9 }
10
11 # A list of integers
12
13 numbers: List[int] = [1, 2, 3, 4, 5, 6]
14
15 # A list that holds dicts that each hold
16 # a string key / int value
17
18 list_of_dicts: List[Dict[str, int]] = [
19     {"key1": 1},
20     {"key2": 2}
21 ]
```

Tuples are a little bit special because they let you declare the type of each element separately.

```
1 from typing import Tuple
2
3 my_data: Tuple[str, int, float] = ("Adam", 10, 5.7)
```

You can create aliases for complex types by assigning them to a new name.

9 Reduce Bugs with Static Type Checking

```
1 from typing import List, Tuple
2
3 LatLngVector = List[Tuple[float, float]]
4
5 points: LatLngVector = [
6     (25.91375, -60.15503),
7     (-11.01983, -166.48477),
8     (-11.01983, -166.48477)
9 ]
```

Sometimes your Python functions might be flexible enough to handle several different types or work on any data type. You can use the `Union` type to declare a function that can accept multiple types and you can use the type `Any` to accept anything.

Running a Type Checker

While Python 3.6 gives you this syntax for declaring types, there's absolutely nothing in Python itself that does anything with these type declarations. To enforce type checking, you need to do one of two things:

1. Download the open-source *mypy* type checker and run it as part of your development workflow.
2. Use an IDE like *PyCharm* that has built-in type checking.

I'd recommend doing both. *PyCharm* and *mypy* use different type checking implementations and they can each catch things that the other doesn't. You can use *PyCharm* for real-time type checking and then run *mypy* as part of your unit tests as a final verification.

When to Use Type Declarations

This type declaration syntax is very new to Python—it only fully works as of Python 3.6. If you show your typed Python code to another Python developer, there’s a good chance they will think you are crazy and not even believe that the syntax is valid. And the mypy type checker is still under development and doesn’t claim to be stable yet.

So it might be a bit early to go whole hog on this for all your projects. I don’t use any type declarations anywhere else in this book because I didn’t want to confuse people who are new to Python.

But if you are working on a new project where you can make Python 3.6 a minimum requirement, it might be worth experimenting with typing. It has the potential to prevent lots of bugs.

You easily can mix code with and without type declarations. It’s not all-or-nothing. So you can start by declaring types in the places that are the most valuable without changing all your code. And since Python itself won’t do anything with these types at runtime, you can’t accidentally break your program by adding type declarations.