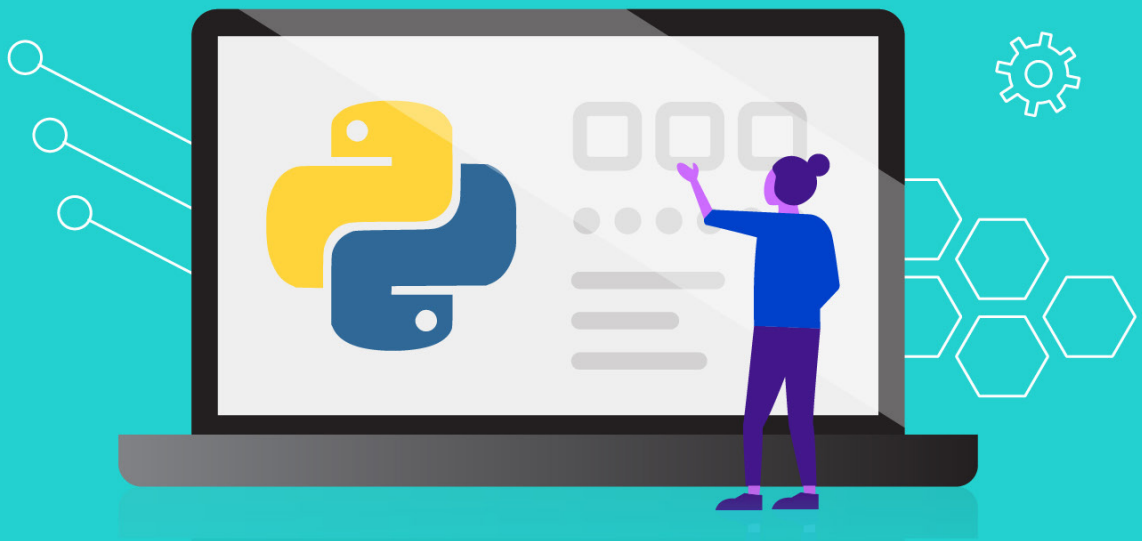
Nylas

Guide to Implementing Python at Scale

Efficiency, innovation, and easier deploys.



2019

About Nylas

Nylas is a developer platform that powers applications with email, calendar, and contacts integrations through a modern REST API. The Nylas API handles more than 100 million API requests per day and has synced more than 15 billion emails. 22,000 developers are signed up to use the API. We're built on Python, and excited to share some of our learnings here.

Overview

The TIOBE index recognized Python as the 2018 programming language of the year, and it's no wonder why. The versatile language can be used for developing desktop GUI applications, APIs, and web applications — to name a few. The friendly syntax of the language makes it easier to maintain your codebase over time as it scales. We at Nylas are excited to share some of the latest Python use cases that are instrumental in laying a strong foundation for your codebase. You can use this knowledge to leverage Python for building apps that scale.

In this guide, you'll learn:

Getting Started with Your First Python Deployment: Git & Pip

Python offers a rich ecosystem of modules. Whether you're building a web server or a machine learning classifier, there's probably a module to help you get started. Today's standardized way of getting these modules is via pip, which downloads and installs from the Python Package Index (aka PyPI). This is just like apt, yum, rubygem, etc.

Most people set up their development environment by first cloning the code using git, and then installing dependencies via pip. So it makes sense why this is also how most people first try to deploy their code. A deploy script might look something like this:

git-pull-pip-install-deploy.sh

```
git clone https://github.com/company/somerepo.git
cd /somerepo
pip install -r requirements.txt
python start_server.py
```

But when deploying large production services, this strategy breaks down for several reasons:

1. Pip does not offer a "revert deploy" strategy.

Running `pip uninstall` doesn't always work properly, and there's no way to rollback to a previous state. Virtualenv could help with this, but it's really not built for managing a history of environments.

2. Installing dependencies with pip can make deploys painfully slow.

Calling `pip install` for a module with C extensions will often build it from source, which can take on the order of minutes to complete for a new virtualenv. Deploys should be a fast lightweight process, taking on the order of seconds.

3. Building your code separately on each host can cause consistency issues.

When you deploy with pip, the version of your app running is not guaranteed to be the same from server to server. Errors in the build process or existing dependencies result in inconsistencies that are difficult to debug, and deploys will fail if the PyPI or your git servers are down.

`pip install` and `git pull` oftentimes depend on external servers. You can choose to use third-party systems (e.g. Github, PyPI) or set up your own servers. Regardless, it is important to make sure that your deploy process meets the same expectations of uptime and scale. Often external services are the first to fail when you scale your own infrastructure, especially with large deployments.

If you're running an app that people depend on and running it across many servers, then the git+pip strategy will only cause headaches. What you need is a deploy strategy that's fast, consistent and reliable.

More specifically:

1. Capability to build code into a single, versioned artifact
2. Unit and system tests that can test the versioned artifact
3. A simple mechanism to cleanly install/uninstall artifacts from remote hosts

Having these three things would allow you to spend more time building features, and less time shipping our code in a consistent way. Using Docker will add complexity to your runtime. Using PEX will add complexity to your build. So, What's the best solution? dh-virtualenv.

Packages: the original "containers".

A couple of years ago, Spotify quietly released a tool called dh-virtualenv, which you can use to build a Debian package that contains a virtualenv.

Building with dh-virtualenv simply creates a Debian package that includes a virtualenv, along with any dependencies listed in the requirements.txt file. When this Debian package is installed on a host, it places the virtualenv at `/usr/share/python/<project-name>`. That's it.

Using a continuous integration server (Jenkins) that runs dh-virtualenv to build the package, and uses Python's wheel cache to avoid re-building dependencies is one of the most efficient ways to deploy code. This creates a single bundled artifact (a Debian package), which is then run through extensive unit and system tests. If the artifact passes, it is certified as safe for prod and uploaded to s3.

A key part of this process is that you can minimize the complexity of your deploy script by leveraging Debian's built-in package manager, dpkg.

A deploy script might look something like this:

```
temp=$(mktemp /tmp/deploy.deb.XXXXX)
curl "https://artifacts.nylas.net/sync-engine-3k48dls.deb" -o
$temp
dpkg -i $temp
sv reload sync-engine
```

To rollback, you can simply deploy the previous versioned artifact. The dpkg utility handles cleaning up the old code for free.

One of the most important aspects of this strategy is that it achieves consistency and reliability, but *still matches your development environment*. If you already use virtualenvs, then you can think of dh-virtualenv as really just a way to ship artifacts to remote hosts. If you choose Docker or PEX, you will have to dramatically change the way you develop locally and introduce a lot of complexity. This helps to simplify the process when working with open source code.

You can ship all of your Python code with Debian packages and your entire codebase (with dozens of dependencies) will take fewer than 2 minutes to build, and seconds to deploy.

Getting started with dh-virtualenv.

If you are experiencing painful Python deployments, then ask your doctor about dh-virtualenv. It might be right for you!

Configuring Debian packages can be tricky for newcomers, here's a handy utility to help you get started called make-deb. It generates a Debian configuration based on the setup.py file in your Python project.

First, install the `make-deb` tool, then run it from the root of your project:

```
cd /my/project
pip install make-deb
make-deb
```

If information is missing from your setup.py file, `make-deb` will ask you to add it. Once it has all the needed details, `make-deb` creates a Debian directory at the root of your project that contains all the configuration you'll need for dh-virtualenv.

Building a Debian package requires you to be running Debian with dh-virtualenv installed. If you're not running Debian, we recommend Vagrant+Virtualbox to set up a Debian VM on Mac or Windows. You can see an example of this configuration by looking at the Vagrantfile in our sync engine git repository.

Finally, running `dpkg-buildpackage -us -uc` will create the Debian package. You don't need to call dh-virtualenv directly, because it's already specified in the configuration rules that `make-deb` created for you. Once this command is finished, you should have a shiny build artifact ready for deployment!

A simple deploy script might look like this:

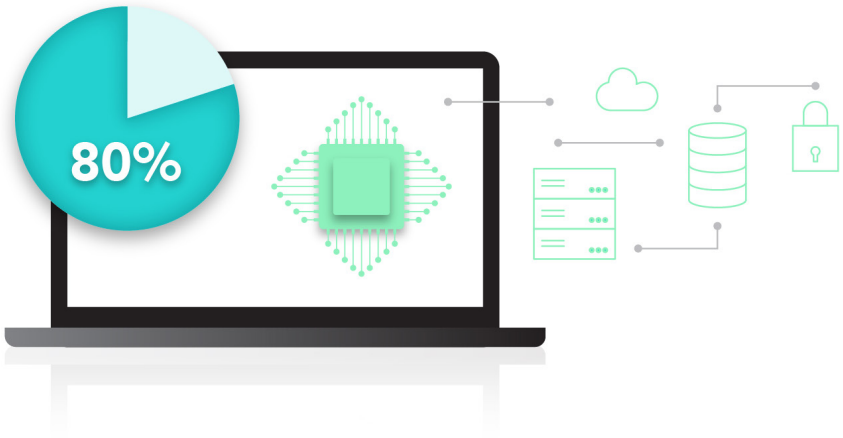
```
scp my-package.deb remote-host.example.org:
ssh remote-host.example.org

# Run the next commands on remote-host.example.org
dpkg -i my-package.deb

/usr/share/python/myproject/bin/python
>>> import myproject # it works!
```

To deploy, you need to upload this artifact to your production machine. To install it, just run `dpkg -i my-package.deb`. Your virtualenv will be placed at `/usr/share/python/<project-name>` and any script files defined in your setup.py will be available in the accompanying `bin` directory. And that's it! You're on your way to simpler deploys.

When building large systems, the engineering dilemma is often to find a balance between creating proper tooling, but not constantly re-architecting a new system from scratch. Using Debian package-based deploys is a great solution for deploying Python apps, and most importantly it lets you ship code faster with fewer issues.

Profiling Python in Products

How to Reduce CPU Usage by 80% Through Python Profiling



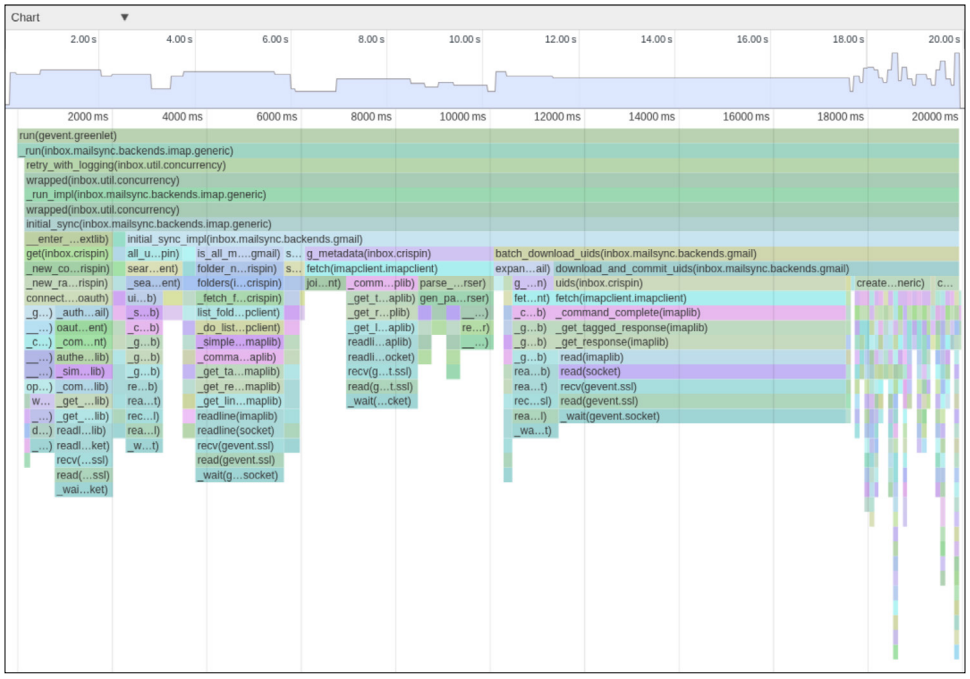
You can reduce CPU usage across your fleet by 80% by using a lightweight profiling strategy that you can run in production.

If you can't measure it, you can't manage it.

CPU optimization starts with measurement and instrumentation, but there's more than one way to profile. It's important to be able to profile both at a small scale using test benchmarks, and at a large scale in a live environment.

To use an example from Nylas, when our API syncs email data with a developer's software application, it's essential that our system analyzes and attributes the data as quickly as possible. To optimize this, we needed to understand how different sync strategies and optimizations affect the first few seconds of a sync.

At Nylas (see example below), we learned that by setting up a local test and adding a bit of custom instrumentation, you can build up a call graph of the program. This is simply a matter of using `sys.setprofile()` to intercept function calls. To better see exactly what's happening, you can export this call graph to a specific JSON format, and then load this into the powerful visualizer built into the Chrome Developer Tools. This lets you inspect the precise timeline of execution:



An example of a call graph exported to JSON and loaded into Chrome Developer Tools by the Nylas team. This visualization allows us to inspect the precise timeline of execution.

Doing it live.

This strategy works well for detailed benchmarking of specific parts of an application, but is poorly suited to analyzing the aggregate performance of a large-scale system. Function call instrumentation introduces significant slowdown and generates a huge amount of data, so you can't just directly run this profiler in production.

However, it's difficult to accurately recreate production slowness in artificial benchmarks, especially when data being synced can have a heterogeneous workload. If the tests aren't actually representative of real-world workload, you'll end up with ineffective optimizations.

The answer to these shortcomings is to add lightweight instrumentation that you can continuously run in your full production cluster, and design a system to roll up the resulting data into a manageable format and size.

At the heart of this strategy is a simple statistical profiler – code that periodically samples the application call stack, and records what it's doing. This approach loses some granularity and is non-deterministic. But this overhead is low and controllable (just choose the sampling interval). Coarse sampling is fine, because you are trying to identify the biggest areas of slowness.

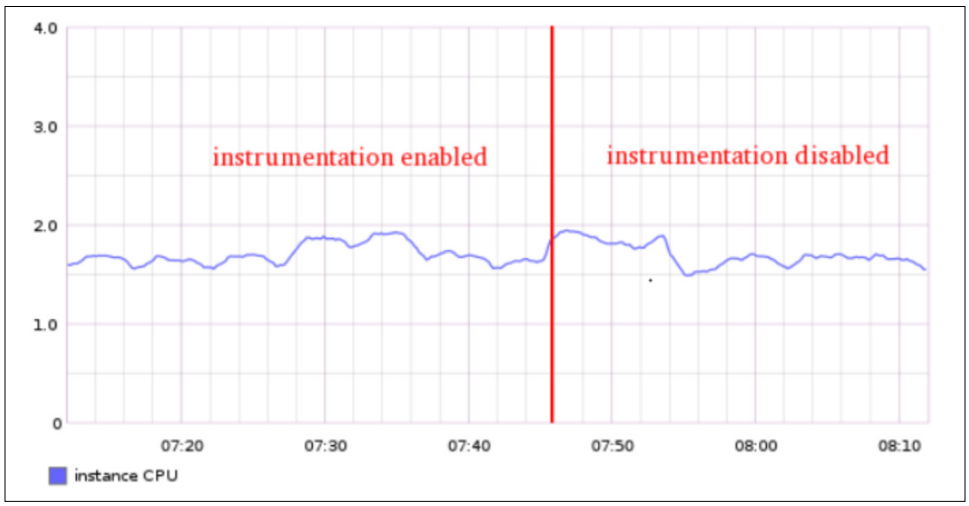
A number of libraries implement variants of this, but in Python, you can write a stack sampler in less than 20 lines:

```
import collections
import signal

class Sampler(object):
   def __init__(self, interval=0.001):
        self.stack_counts = collections.defaultdict(int)
        self.interval = interval

    def _sample(self, signum, frame):
       stack = []
        while frame is not None:
            formatted_frame = '{}({})'.format(frame.f_code.co_name,
                                              frame.f_globals.
                                              get('__name__'))
            stack.append(formatted_frame)
            frame = frame.f_back

        formatted_stack = ';'.join(reversed(stack))
        self.stack_counts[formatted_stack] += 1
        signal.setitimer(signal.ITIMER_VIRTUAL, self.interval, 0)

    def start(self):
        signal.signal(signal.VTALRM, self._sample)
        signal.setitimer(signal.ITIMER_VIRTUAL, self.interval, 0)
```

Calling `Sampler.start()` sets the Unix signal ITIMER_VIRTUAL to be sent after the number of seconds specified by `interval`. This essentially creates a repeating alarm that will run the `_sample` method.

When the signal fires, this function saves the application's stack, and keeps track of how many times you have sampled that same stack. Frequently sampled stacks correspond to code paths where the application is spending a lot of time.

The memory overhead associated with maintaining these stack counts stays reasonable, since the application only executes so many different frames. It's also possible to bound the memory usage, if necessary, by periodically pruning infrequent stacks. In our application, the actual CPU overhead is demonstrably negligible:



This graph of memory usage before and after the deploy enabling the profiler does not reflect a significant increase in CPU usage.

Now that you have added instrumentation in the application, you can have each worker process expose its profiling data via a simple HTTP interface (see code). This lets you take a production worker process and generate a flamegraph that concisely illustrates where the worker is spending time:

```
curl $host:$port | flamegraph.pl > profile.svg
```

The longer the bar, the more time that code block takes to execute. You can figure out the most time-inefficient functions, and make them more performant.

This visualization makes it easy to quickly spot where CPU time is being spent in the actual process. For example, around 15% of runtime is being spent in the `get()` method highlighted above, executing a database load that turns out to normally be unnecessary. This wasn't evident in local testing, but now it's easy to identify and fix.

However, the load on any single worker process isn't necessarily representative of the aggregate workload across all processes and instances. You want to be able to aggregate stacktraces from multiple processes. You also need a way to save historical data, as the profiler only stores traces for the current lifetime of the process. To do this, you can run a collector agent that periodically polls all sync engine processes (across multiple machines), and persists the aggregated profiling data to its own local store.

Finally, a lightweight web app can visualize this data on demand. Answering the question, "Where is your application spending CPU time?" is now as simple as visiting an internal URL:

We hosted the flame graph of our profiling data on an internal application, so our developers could easily view and track changes over time.

Since you can render profiles for any given time interval, it's easy to track down the cause of any regressions and the moment they were introduced.

Monitoring your application.

Being able to measure and introspect about your services in a variety of ways is crucial to keeping them stable and performant. The simple tooling presented here can be a vital part of a larger monitoring infrastructure for your application.

4 Python Libraries You Can't Live Without

As we all know, open-source code isn't just developed and left standing; it's regularly maintained and extended by talented and dedicated Python developers.

These crucial libraries will save you from reinventing the wheel time and time again, and you can get a lot of mileage out of them.

1 ptpython for fast & easy internal tooling

The ptpython REPL brings autocomplete, syntax highlighting and multiline editing to the Python shell, creating a robust and customizable Python environment. You can use it to enhance your Python console with pre-loaded text, helper functions and objects for our developers and our developer success engineers to use. If an error pops up in our logs, you can quickly load the buggy object in a Python REPL, debug the root cause, and experiment with solutions directly in the Python shell.

2 tldextract for parsing URLs

If you've ever tried to write a regular expression to define a valid URL, you know it's not as straightforward as it sounds. The normal `.com`, `.org` and `.net` top-level domains are easy enough to codify, but these days `.limo`, `.pizza` and `.duck` are all valid suffixes as well. If you have a need to parse URLs to validate emails and server addresses, you use tldextract to separate the protocol from the domain, subdomain(s) and suffix in a given URL.

3 vcr for fast, reusable test fixtures

If you work with a lot of APIs, this means you likely make multiple HTTP requests. To simplify and speed up testing, you can use vcr to record the responses from the HTTP interactions in your test code and save them to a flat file in your codebase, called a cassette.

After the initial response has been recorded, it is "replayed" by later tests, so you don't have to make live requests to external APIs in your test code. This makes your tests fast (no real HTTP requests anymore), deterministic (the test will continue to pass, even if you are offline) and accurate (the response will contain the same headers and body you get from a real request).

4 expiringdict for ordered caches

Mailgun's expiringdict is a great example of a library that does one thing, and does it really well. Their data structure, based off collections' `OrderedDict`, creates a dictionary with a max length, and items that "expire" after a certain amount of time, to be used as a cache.

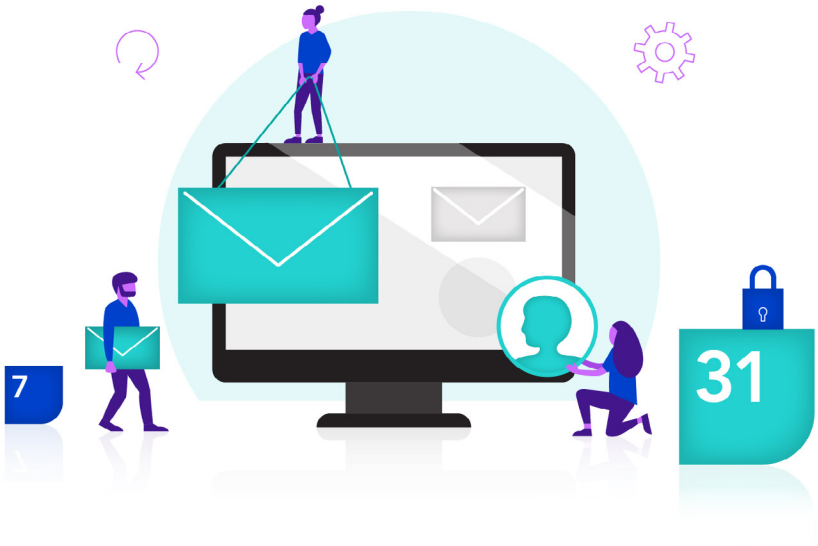
Caching the results of queries you know are executed frequently lets you serve up faster results, with less load on the database. To prevent the cache from getting too big, which impacts performance, you can cap the size, and it's convenient to set an expiration date on the data, so that your cache doesn't get stale.

Bringing It Together

As you can see, Python can be used for a variety of functions that aid with building a reliable product in an efficient manner. With its robust libraries, your Python code will be readable, maintainable, and compatible with other major platforms and systems as you scale. Using Python will also simplify complex development issues and make it easy to implement vital testing measures.

See what Python can do for you, and share your learnings with us on Twitter: @Nylas!

Learn more about the Nylas APIs for email, calendar and contacts, or create a free developer account.



Nylas

Nylas.com

Github.com/Nylas

@Nylas