[Pipenv: A Guide to the New Python Packaging Tool – Real Python](https://realpython.com/pipenv-guide/)

Pipenv là một công cụ "Đóng gói" dành cho Python, giải quyết một số vấn đề thường gặp liên quan đến quy trình làm việc đặc trưng sử dụng pip, virtualenv, và requirements.txt.

Ngoài việc giải quyết một số vấn đề thường gặp, nó hợp nhất và đơn giản hoá quy trình phát triển với một câu lệnh duy nhất.

Hướng dẫn này sẽ đi qua các vấn đề mà Pipenv giải quyết và làm cách nào để quản lý các gói package, thư viện với Pipenv. Ngoài ra, nó sẽ bao gồm cách Pipenv phù hợp với các phương pháp trước đó.

**Problems that Pipenv Solves**

Để hiểu được lợi ích của PIPENV, điều quan trọng là phải đi qua các phương pháp hiện tại để quản lý package và dependency củaPython.

Hãy bắt đầu với một tình huống điển hình của việc xử lý các gói của bên thứ ba. Sau đó, chúng tôi sẽ xây dựng theo cách của chúng tôi để triển khai một ứng dụng Python hoàn chỉnh.

**Dependency Management with requirements.txt**

Hãy tưởng tượng bạn đang làm việc trên một dự án Python sử dụng package của bên thứ ba như Flask. Bạn sẽ cần chỉ định yêu cầu đó để các nhà phát triển và hệ thống tự động khác có thể chạy ứng dụng của bạn.

Vì vậy, bạn quyết định đưa phụ thuộc bình vào file requirements.txt: flask

Tuyệt vời, mọi thứ hoạt động tốt trên máy của bạn và sau khi hack trên ứng dụng của bạn một lúc, bạn quyết định chuyển nó vào sản xuất. Đây là lúc mọi thứ bắt đầu có một chút đáng sợ…

File requirements.txt không chỉ định phiên bản của flask để sử dụng. Trong trường hợp này, Pip Install -r requirements.txt sẽ mặc định cài đặt phiên bản mới nhất. Điều này ổn trừ khi có giao diện hoặc thay đổi trong phiên bản mới nhất phá vỡ ứng dụng của bạn.

Giả sử một phiên bản mới của Flask đã được phát hành. Tuy nhiên, nó không tương thích ngược với phiên bản bạn đã sử dụng trong quá trình phát triển.

Bạn biết rằng phiên bản của Flask bạn đã sử dụng trong quá trình phát triển hoạt động tốt. Vì vậy, để sửa chữa mọi thứ, bạn cố gắng cụ thể hơn một chút trong file requirements.txt của mình. Bạn thêm một phiên bản xác định vào flask. Điều này cũng được gọi là ghim một dependency: flask==0.12.1

Pinning the flask dependency to a specific version ensures that a pip install -r requirements.txt sets up the exact version of flask you used during development. But does it really?

Hãy nhớ rằng bản thân Flask cũng có dependency (PIP tự động cài đặt). Tuy nhiên, bản thân Flask không chỉ định các phiên bản chính xác cho các phụ thuộc của nó. Ví dụ: nó cho phép bất kỳ phiên bản nào của Werkzeug> = 0,14.

Một lần nữa, giả sử một phiên bản mới của Werkzeug đã được phát hành, nó gây ra một lỗi về hiển thị cho ứng dụng của bạn. Lần này, khi bạn thực hiện install -r requirements.txt, bạn sẽ nhận được Flask==0.12.1 vì bạn đã ghim yêu cầu đó. Tuy nhiên, thật không may, bạn sẽ nhận được phiên bản mới nhất của Werkzeug. Một lần nữa, your product breaks in production.

Vấn đề thực sự ở đây là bản dựng không xác định. Ý tôi là, được đưa ra cùng một đầu vào (tệp requirements.txt), PIP không phải lúc nào cũng tạo ra cùng một môi trường. Hiện tại, bạn không thể dễ dàng sao chép môi trường chính xác mà bạn có trên máy phát triển trong sản xuất.

Chạy pip freeze sẽ tạo ra một loạt các dependency mà bạn có thể add vào requirements.txt:

click==6.7

Flask==0.12.1

itsdangerous==0.24

Jinja2==2.10

MarkupSafe==1.0

Werkzeug==0.14.1

Với những dependency này, bạn có thể đảm bảo các package được cài trong môi trường sản xuất của bạn khớp với môi trường phát triển, app của bạn sẽ ko bất ngờ bị break. Nhưng phương pháp này thật ko may lại dẫn tới một loạt các vấn đề mới.

Bây giờ bạn đã chỉ định các phiên bản chính xác của mỗi gói của bên thứ ba, bạn có trách nhiệm giữ các phiên bản này được cập nhật, kể cả dependency của Flask. Điều gì sẽ xảy ra nếu có một lỗi bảo mật được phát hiện trong werkzeug == 0.14.1 rằng các nhà bảo trì package ngay lập tức sửa nó trong werkzeug==0.14.2? Bạn thực sự cần cập nhật lên werkzeug==0.14.2 để tránh mọi vấn đề bảo mật phát sinh từ Werkzeug.

Đầu tiên, bạn cần lưu ý rằng có một vấn đề với phiên bản bạn có. Sau đó, bạn cần có được phiên bản mới trong môi trường sản xuất của mình trước khi ai đó khai thác lỗ hổng bảo mật. Vì vậy, bạn phải thay đổi requirements.txt theo cách thủ công để chỉ định phiên bản mới werkzeug == 0.14.2. Như bạn có thể thấy trong tình huống này, trách nhiệm cập nhật với các bản cập nhật cần thiết thuộc về bạn.

Sự thật là bạn thực sự không quan tâm đến phiên bản Werkzeug nào được cài đặt miễn là nó không phá vỡ hoạt động của chương trình. Trên thực tế, bạn có thể muốn phiên bản mới nhất đảm bảo rằng bạn có các bug fixes, các bản vá bảo mật, tính năng mới, tối ưu hóa hơn, v.v.

Câu hỏi thực sự là: "**Làm thế nào để bạn cho phép các bản dựng xác định cho dự án Python của bạn mà không có trách nhiệm cập nhật các phiên bản sub-dependencies?**"

**Development of Projects with Different Dependencies**

Hãy nói một chút về một vấn đề phổ biến khác phát sinh khi bạn đang làm việc trên nhiều dự án. Hãy tưởng tượng rằng Projecta cần django==1.9, nhưng ProjectB cần django==1.10.

Theo mặc định, Python cố gắng lưu trữ tất cả các gói của bên thứ ba của bạn ở một vị trí toàn hệ thống. Điều này có nghĩa là mỗi khi bạn muốn chuyển đổi giữa ProjectA và ProjectB, bạn phải đảm bảo phiên bản phù hợp của Django được cài đặt. Điều này làm cho việc chuyển đổi giữa các dự án phiền phức vì bạn phải gỡ cài đặt và cài đặt lại các gói để đáp ứng các yêu cầu cho mỗi dự án.

Giải pháp tiêu chuẩn là sử dụng một môi trường ảo có lưu trữ gói bên thứ ba và thực thi Python của riêng mình. Bằng cách đó, ProjectA và ProjectB được phân tách hoàn toàn. Bây giờ bạn có thể dễ dàng chuyển đổi giữa các dự án vì chúng không chia sẻ cùng một vị trí lưu trữ gói. PackageA có thể có bất kỳ phiên bản django nào mà nó cần trong môi trường riêng của nó và PackageB có thể có những gì nó cần hoàn toàn riêng biệt. Một công cụ rất phổ biến cho điều này là VirtualEnv (hoặc VENV trong Python 3).

Pipenv có trình quản lý môi trường ảo tích hợp để bạn có một công cụ duy nhất để quản lý gói của mình.

**Dependency Resolution**

Giả sử bạn có 2 packageA và packageB trong file requiements.txt. packageA có một sub-dependency là packageC>=1.0. Phía bên kia, packageB có cùng một sub-dependency nhưng yêu cầu packageC <=2.0.

Lý tưởng nhất là khi bạn cố gắng cài đặt packageA và packageB, công cụ cài đặt sẽ xem xét các yêu cầu cho packageC (là> = 1.0 và <= 2.0) và chọn một phiên bản đáp ứng các yêu cầu đó. Bạn sẽ hy vọng rằng công cụ sẽ giải quyết các phụ thuộc để chương trình của bạn hoạt động tốt. This is what I mean by “dependency resolution.”

Thật không may, bản thân PIP chưa có giải pháp thực sự tại thời điểm này, nhưng có một vấn đề mở để hỗ trợ nó. Cách PIP sẽ xử lý kịch bản trên như sau:

1. Nó cài đặt packageA và tìm một version cho packageC đáp ứng yêu cầu đầu tiên: packageC>=1.0
2. Pip sau đó cài đặt phiên bản mới nhất của packageC để đáp ứng yêu cầu packageC<=2.0. Giả sử phiên bản mới nhất của packageC là 3.1

Đây là lúc rắc rối tiềm ẩn bắt đầu. Nếu version packageC được chọn bởi pip ko phù hợp với các yêu cầu sau đó (VD packageC<=2.0), cài đặt sẽ Fail.

"Giải pháp" cho vấn đề này là chỉ định phạm vi cần thiết cho sub-dependency (packageC) trong file requirements.txt. Bằng cách đó, PIP có thể giải quyết xung đột này và cài đặt gói đáp ứng các yêu cầu đó:

packageC>=1.0,<=2.0

packageA

packageB

Giống như trước, nếu như packageA thay đổi yêu cầu về sub-dependency mà bạn ko biết, yêu cầu đầu tiên của bạn cho packageC có thể ko còn được chấp nhận và việc cài đặt có thể lại thất bại. Một lần nữa, bạn lại phải có trách nhiệm cập nhật các sub-dependencies.

Lý tưởng nhất, có một công cụ cài đặt đủ thông minh để cài đặt các gói đáp ứng tất cả các yêu cầu mà không cần bạn chỉ định rõ ràng các phiên bản sub-deependency.

**Pipenv Introduction**

Bây giờ chúng ta hãy xem Pipenv giải quyết công việc này như thế nào.

$ pip install pipenv

Một khi bạn đã cài nó, bạn có thể quên pip vì về cơ bản, Pipenv hoạt động như một sự thay thế. Nó cũng đưa ra 2 file mới. **Pipfile** (thay thế cho requirements.txt) và Pipfile.lock (cho phép xây dựng bản dựng xác định).

Pipenv sử dụng pip và virtualenv nhưng đơn giản hoá việc sử dụng với một câu lệnh duy nhất.

**Example Usage**

Hãy bắt đầu lại với việc tạo ứng dụng Python tuyệt vời của bạn. Đầu tiên, sinh ra một shell trong một virtualenv (hay một project) để cô lập sự phát triển của ứng dụng này:

$ pipenv shell

Điều này sẽ tạo ra một virtual environment nếu nó chưa tồn tại. Pipenv tạo tất cả các môi trường ảo của bạn ở một vị trí mặc định. Nếu bạn muốn thay đổi hành vi mặc định này của Pipenv, there are some [environmental variables for configuration](https://docs.pipenv.org/advanced/#configuration-with-environment-variables).

Bạn có thể buộc tạo ra môi trường Python 2 hoặc 3 với các đối số -two và --three tương ứng. Nếu không, Pipenv sẽ sử dụng bất kỳ virtualenv mặc định nào tìm thấy.

*Sidenote: Nếu bạn yêu cầu một phiên bản Python cụ thể hơn, bạn có thể cung cấp đối số - -python với phiên bản bạn yêu cầu. Ví dụ: - -Python 3.6*

Bây giờ bạn có thể cài đặt gói bên thứ 3 bạn cần, Flask. Ồ, nhưng bạn biết rằng bạn cần phiên bản 0.12.1 chứ không phải phiên bản mới nhất, vì vậy hãy tiếp tục và cụ thể: **pipenv install flask==0.12.1**. Bạn sẽ thấy như thế này trong terminal của mình:

Shell 
Adding to Pipfile's [packages] ... 
Pipfile.lock not found, creating... 

If you want to install something directly from a version control system (VCS), you can! You specify the locations similarly to how you’d do so with pip. For example, to install the requests library from version control, do the following:

$ pipenv install -e git+https://github.com/requests/requests.git#egg=requests

Note the -e argument above to make the installation editable. Currently, this is required for Pipenv to do sub-dependency resolution.

Giả sử bạn cũng có một số thử nghiệm đơn vị cho ứng dụng tuyệt vời này và bạn muốn sử dụng pytest để chạy chúng. Bạn không cần pytest trong sản xuất, bạn có thể chỉ định rằng gói này chỉ dành cho phát triển với argument **--dev**:

$ pipenv install pytest --dev

Cung cấp "--dev" ​​sẽ đặt dependency vào một vị trí [dev-packages] đặc biệt trong Pipfile. Các gói phát triển này chỉ được cài đặt nếu bạn chỉ định tham số **--dev** với **pipenv install**.

Các phần dependency chỉ cần cho phát triển tách biệt với các các dependency cần thiết cho based code hoạt động. Thông thường, điều này sẽ được thực hiện với các tệp bổ sung như dev-requirements.txt hoặc test-rquirements.txt. Bây giờ, mọi thứ được hợp nhất trong một Pipfile duy nhất theo các phần khác nhau.

Được rồi, giả sử rằng bạn đã có mọi thứ làm việc trong môi trường phát triển local của mình và bạn đã sẵn sàng để đẩy nó vào sản xuất. Để làm điều đó, bạn cần khóa môi trường của mình để bạn có thể đảm bảo bạn có cùng một sản phẩm trong sản xuất:

$ pipenv lock

Điều này sẽ create/update pipfile.lock của bạn, mà bạn sẽ không bao giờ cần (và không bao giờ có ý định) chỉnh sửa thủ công. Bạn phải luôn luôn sử dụng tệp được tạo.

Bây giờ, một khi bạn nhận được code và pipfile.lock trong môi trường sản xuất của mình, bạn nên cài đặt môi trường ổn định cuối cùng được ghi lại:

$ pipenv install --ignore-pipfile

Lock file cho phép các bản dựng xác định bằng cách chụp lại tất cả các phiên bản của các gói trong môi trường (tương tự như kết quả của việc pip freeze).

Bây giờ hãy nói rằng một nhà phát triển khác muốn thực hiện một số bổ sung cho code của bạn. Trong tình huống này, họ sẽ nhận được code, bao gồm cả Pipfile và sử dụng lệnh này:

$ pipenv install --dev

Điều này cài đặt tất cả các dependencies cần thiết để phát triển, bao gồm cả các dependencies thông thường và những dependencies bạn đã chỉ định với than số --dev trong quá trình cài đặt.

*Khi một phiên bản cụ thể không được chỉ định trong Pipfile, lệnh install cho phép các dependencies (và sub-dependencies) cập nhật phiên bản mới nhất.*

Đây là một lưu ý quan trọng bởi vì nó giải quyết một số vấn đề trước đây chúng ta đã thảo luận. Để chứng minh, giả sử một phiên bản mới của một trong những dependency của bạn đc phát hành. Bởi vì bạn không cần một phiên bản cụ thể của dependency này, bạn không chỉ định phiên bản chính xác trong Pipfile. Khi bạn pipenv install, phiên bản mới của dependency sẽ được cài đặt trong môi trường phát triển của bạn.

Bây giờ bạn thực hiện các thay đổi đối với code và chạy một số thử nghiệm để xác minh mọi thứ vẫn đang hoạt động như mong đợi. . Giống như trước đây, bạn có thể sao chép môi trường mới này trong sản xuất với lock file.

Như bạn có thể thấy từ kịch bản này, bạn không còn phải buộc các phiên bản chính xác mà bạn không thực sự cần để đảm bảo môi trường phát triển và sản xuất của bạn là như nhau. Bạn cũng không cần phải luôn cập nhật sub-dependencies mà bạn "không quan tâm." Quy trình công việc này với Pipenv, kết hợp với thử nghiệm tuyệt vời của bạn, khắc phục các vấn đề thực hiện thủ công tất cả các quản lý dependency của bạn.

**Pipenv’s Dependency Resolution Approach**

**The Pipfile**

**The Pipfile.lock**

**Pipenv Extra Features**

**Package Distribution**

**Yes, I need to distribute my code as a package**

**I don’t need to distribute my code as a package**

**I already have a requirements.txt. How do I convert to a Pipfile?**

Nếu bạn chạy pipenv install, nó sẽ tự động phát hiện requirements.txt và chuyển đổi nó thành Pipfile, output sẽ như sau:

requirements.txt found, instead of Pipfile! Converting…

Warning: Your Pipfile now contains pinned versions, if your requirements.txt did.

We recommend updating your Pipfile to specify the "\*" version, instead.

Nếu bạn đã ghim các phiên bản chính xác trong requirements.txt, có lẽ bạn sẽ muốn thay đổi Pipfile của mình để chỉ định các phiên bản chính xác mà bạn thực sự yêu cầu. Điều này sẽ cho phép bạn đạt được những lợi ích thực sự của việc chuyển đổi. Ví dụ: giả sử bạn có có cái này nhưng không cần phiên bản chính xác đó của Numpy:

[packages]

numpy="==1.14.1"

Nếu bạn không có bất kỳ yêu cầu phiên bản cụ thể nào cho các dependencies của mình, bạn có thể sử dụng ký tự đại diện \* để nói với PipenV rằng bất kỳ phiên bản nào có thể được cài đặt:

[packages]

numpy="\*"

Nếu bạn cảm thấy lo lắng về việc cho phép bất kỳ phiên bản nào với \*, thì đó thường là đặt cược an toàn để chỉ định >= phiên bản bạn đã sử dụng và vẫn có thể tận dụng các phiên bản mới:

[packages]

numpy=">=1.14.1"

Tất nhiên, cập nhật các bản phát hành mới cũng có nghĩa là bạn có trách nhiệm đảm bảo code của bạn vẫn hoạt động như mong đợi khi các package thay đổi. Điều này có nghĩa là một bộ thử nghiệm là điều cần thiết cho toàn bộ luồng pipenv này nếu bạn muốn đảm bảo các bản phát hành chức năng của your code.

Bạn cho phép các gói cập nhật, chạy thử nghiệm của bạn, đảm bảo tất cả chúng đều vượt qua, khóa môi trường của bạn và sau đó bạn có thể nghỉ ngơi dễ dàng khi biết rằng bạn chưa giới thiệu các breaking change. Nếu mọi thứ bị phá vỡ vì sự phụ thuộc, you’ve got some regression tests to write and potentially some more restrictions on versions of dependencies.

Ví dụ: nếu Numpy==1.15 được cài đặt sau khi chạy pipenv install và nó sẽ phá vỡ code của bạn, mà bạn hy vọng sẽ nhận thấy trong quá trình phát triển hoặc trong các thử nghiệm của mình, bạn có một vài lựa chọn:

1. Cập nhật code của bạn để hoạt động với phiên bản mới của package.

Nếu khả năng tương thích ngược với các phiên bản trước của dependency là không thể, bạn cũng sẽ cần phải chỉ định phiên bản cần thiết của mình trong Pipfile:

[packages]

numpy=">=1.15"

1. Hạn chế phiên bản của dependency trong Pipfile là < phiên bản vừa làm hỏng code của bạn:

[packages]

numpy=">=1.14.1, <1.15"

Lựa chọn 1 được ưu tiên vì nó đảm bảo rằng mã của bạn đang sử dụng các gói dependency mới nhất. Tuy nhiên, lựa chọn 2 mất ít thời gian hơn và không yêu cầu thay đổi code, chỉ hạn chế các dependency.

Ngoài ra, bạn có thể đi theo cách khác và tạo các tệp yêu cầu từ Pipfile:

$ pipenv lock -r > requirements.txt  
$ pipenv lock -r -d > dev-requirements.txt

**What’s next?**

**Is Pipenv worth checking out?**

**References, further reading, interesting discussions, and so forth**