Advanced Python Documentation

Release 0.3

Jason McVetta

Sep 27, 2017

Contents

CHAPTER 1

Introduction

This class will familiarize the student with modern tools, libraries, and techniques used by Python programmers.

Classroom delivery of this course is available from Silicon Bay Training, who sponsored its development.

Background Assumptions

- Solid understanding of Python language syntax and conventions. Successful completion of *Python Essentials* or equivalent.
- Basic level familiarity with Linux command line.
- Significant familiarity with web concepts such as HTML syntax, HTTP requests and responses, APIs, etc.

Resources

The Sphinx source code for this curriculum can be found on Github.

The latest version this curriculum is published at Read the Docs. It is also available in PDF and ePub formats.

CHAPTER 2

Tool Setup

System Packages

Since we are starting with a fresh install of Ubuntu 12.04, we need to install various useful libraries and applications. Ubuntu, like most Linux systems, uses a package manager to control installation of software packages. The Ubuntu package manager is called `apt-get`.

If you are using a version of Ubuntu other than 12.04, or a different unix system, the package names will probably be similar but not exactly the same.

Package Descriptions

The following packages are required to complete this course:

Package	Description
openjdk-7-jdk	Java Development Kit
openssh-server	SSH server
postgresql-9.1	Database Server
postgresql-server-dev-9.1	Development headers for PostgreSQL
python-pip	Python package manager
python-virtualenv	Python virtual environment support
python2.7-dev	Python development headers
virtualenvwrapper	Tool for convenient virtual environment usage

Installation

Use `apt-get install` to install the required packages, as well as their dependencies:

```
$ sudo apt-get install \
openjdk-7-jdk \
openssh-server \
```

```
postgresql-9.1 \
postgresql-server-dev-9.1 \
python-pip \
python-virtualenv \
python2.7-dev \
virtualenvwrapper
```

Virtual Environments

A virtual environment is a local Python environment isolated from the system-wide environment.

virtualenv

The term "virtualenv" can refer to the command `virtualenv`, used to create a virtual environment, or to the virutal environment itself.

virtualenvwrapper

`virtualenvwrapper` is a set of extensions to the `virtualenv` tool. The extensions include wrappers for creating and deleting virtual environments and otherwise managing your development workflow, making it easier to work on more than one project at a time without introducing conflicts in their dependencies[1].

```
$ mkvirtualenv class
New python executable in class/bin/python
Installing distribute.....................................................................
↪.....................................................................................
↪.....................................done.
Installing pip...............done.
virtualenvwrapper.user_scripts creating /home/jason/.virtualenvs/class/bin/
↪predeactivate
virtualenvwrapper.user_scripts creating /home/jason/.virtualenvs/class/bin/
↪postdeactivate
virtualenvwrapper.user_scripts creating /home/jason/.virtualenvs/class/bin/preactivate
virtualenvwrapper.user_scripts creating /home/jason/.virtualenvs/class/bin/
↪postactivate
virtualenvwrapper.user_scripts creating /home/jason/.virtualenvs/class/bin/get_env_
↪details
(class)$ pip freeze # A few packages are installed by default
argparse==1.2.1
distribute==0.6.24
wsgiref==0.1.2
```

Note that when the virtual environment is active, its name (in this case "class") is prepended to the shell prompt:

```
$ # Ordinary shell prompt
(class)$ # Virtual environment "class" is active
```

If later you have logged out, and want to activate this virtual environment, you can use the `workon` command:

```
$ workon class
(class)$
```

[1] http://www.doughellmann.com/projects/virtualenvwrapper/

You can deactivate the virtual environment with the `deactivate` command:

```
(class)$ deactivate
$ # Back to normal shell prompt
```

Location of Virtualenvs

By default, `virtualenvwrapper` stores your virtual environments in `~/.virtualenvs`. However you can control this by setting the `WORKON_HOME` environment variable. This could potentially be used for shared virtual environments, perhaps with group write permission.

```
export WORKON_HOME=/path/to/virtualenvs
```

Virtual Environments for Scripts

There are several ways you can run scripts that rely on a virtualenv:

- Use Fabric's `prefix()` context manager when calling the script remotely:

```
def task():
    with prefix('workon class'):
        run('uptime')
        run('uname -a')
```

- Have whatever is calling your script (`cron` etc) call `workon` first.
- Specify your virtualenv's Python interpreter directly in the script's bangline.
- Use a bash script as a wrapper. Ugly, but sometimes convenient.

Python Packages

Earlier we installed packages system-wide using the Ubuntu package manager, `apt-get`. Now we will install some Python packages locally, into our virtual environment. We will use Pip, the Python package manager. Pip is aware of virtual environments. If you have a virtual environment active when you call Pip, the requested packages will be installed into the active virtual environmentment.

Package Descriptions

The following packages are required to complete this course:

Package	Description
argparse	Command-line parsing library
django	Web application framework
fabric	SSH library and command line tool
ipython	Interactive Python shell
psycopg2	PostgreSQL driver
requests	HTTP for humans

Installation

Use `pip install` to install the required packages, as well as their dependencies:

```
(class)$ pip install \
argparse \
django \
fabric \
ipython \
psycopg2 \
requests \
```

Aptana Studio

Aptana Studio is an IDE - integrated development environment - based on the Eclipse framework. It provides powerful tools for exploring, understanding, and refactoring your code.

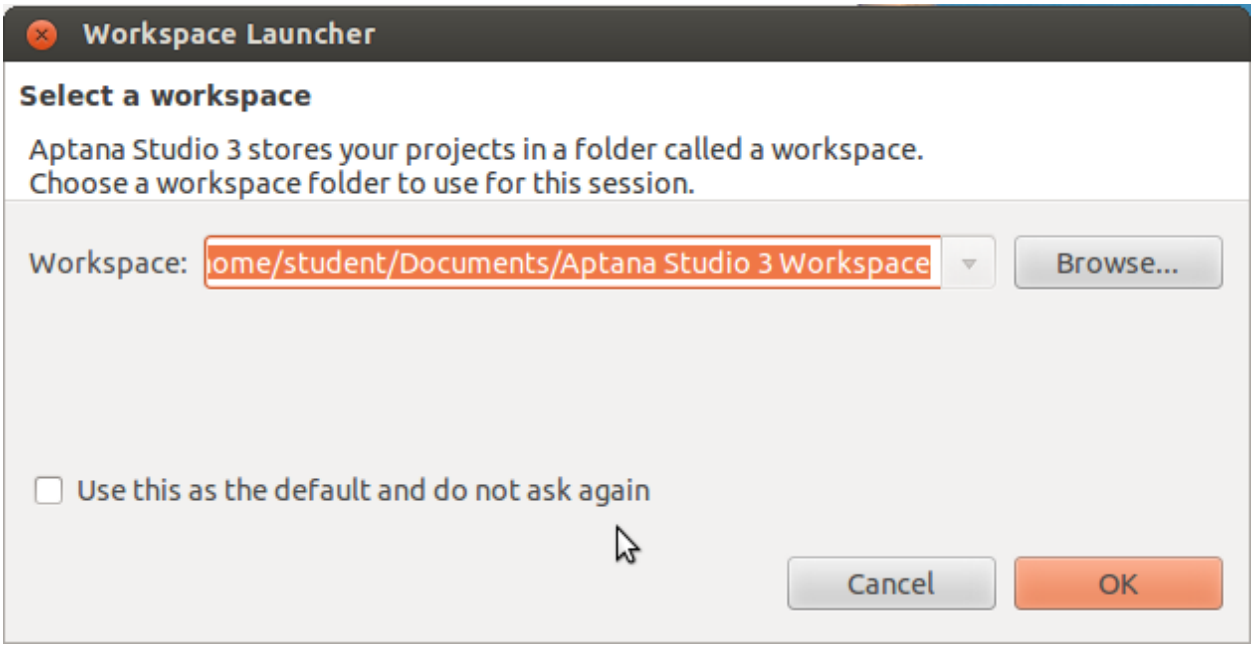
Because Aptana Studio is Eclipse + a plugin, in class I may refer to "Aptana" and "Eclipse" interchangeably. Unless explicitly noted, both terms refer to the combination of Eclipse framework + Aptana Studio plugin.

Aptana's Python support was formerly a separate Eclipse plugin called *PyDev*. PyDev was purchased by Aptana and folded into Aptana Studio. Aptana can be installed as a seperate download, or as an Eclipse plugin. For convenience we will download the whole application.

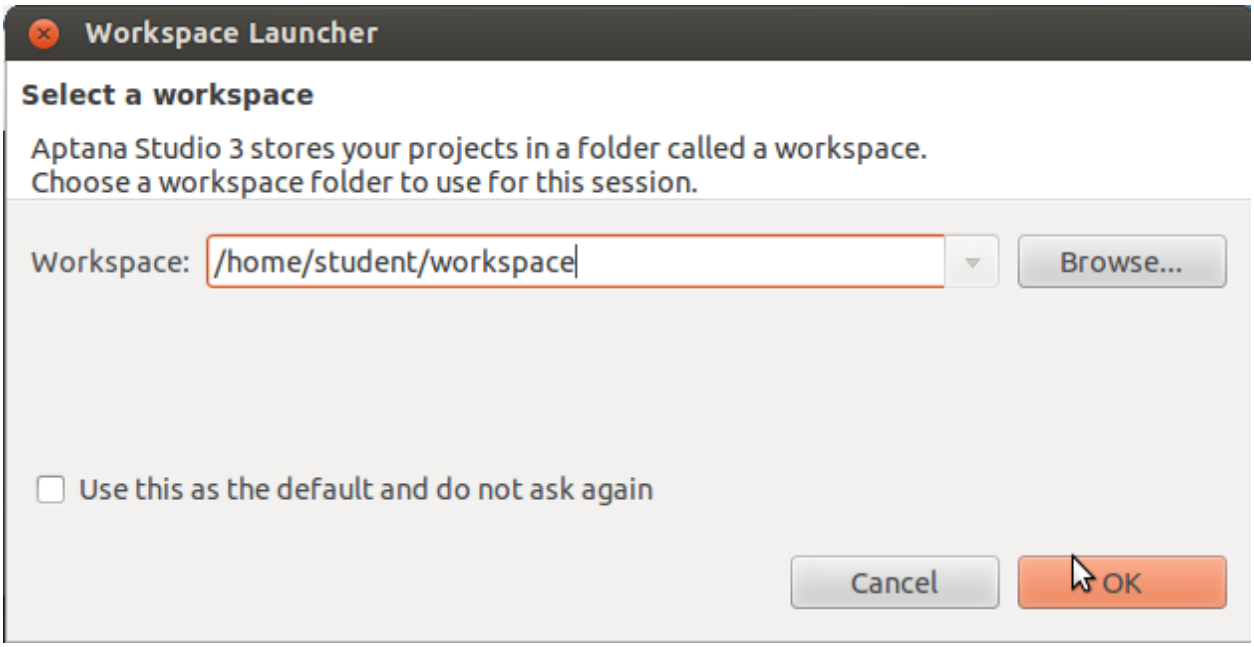
> http://aptana.com/products/studio3/download

Workspace

When it starts up, Aptana will ask you what folder you want to use as a workspace. The default is not very good, as its name contains spaces:

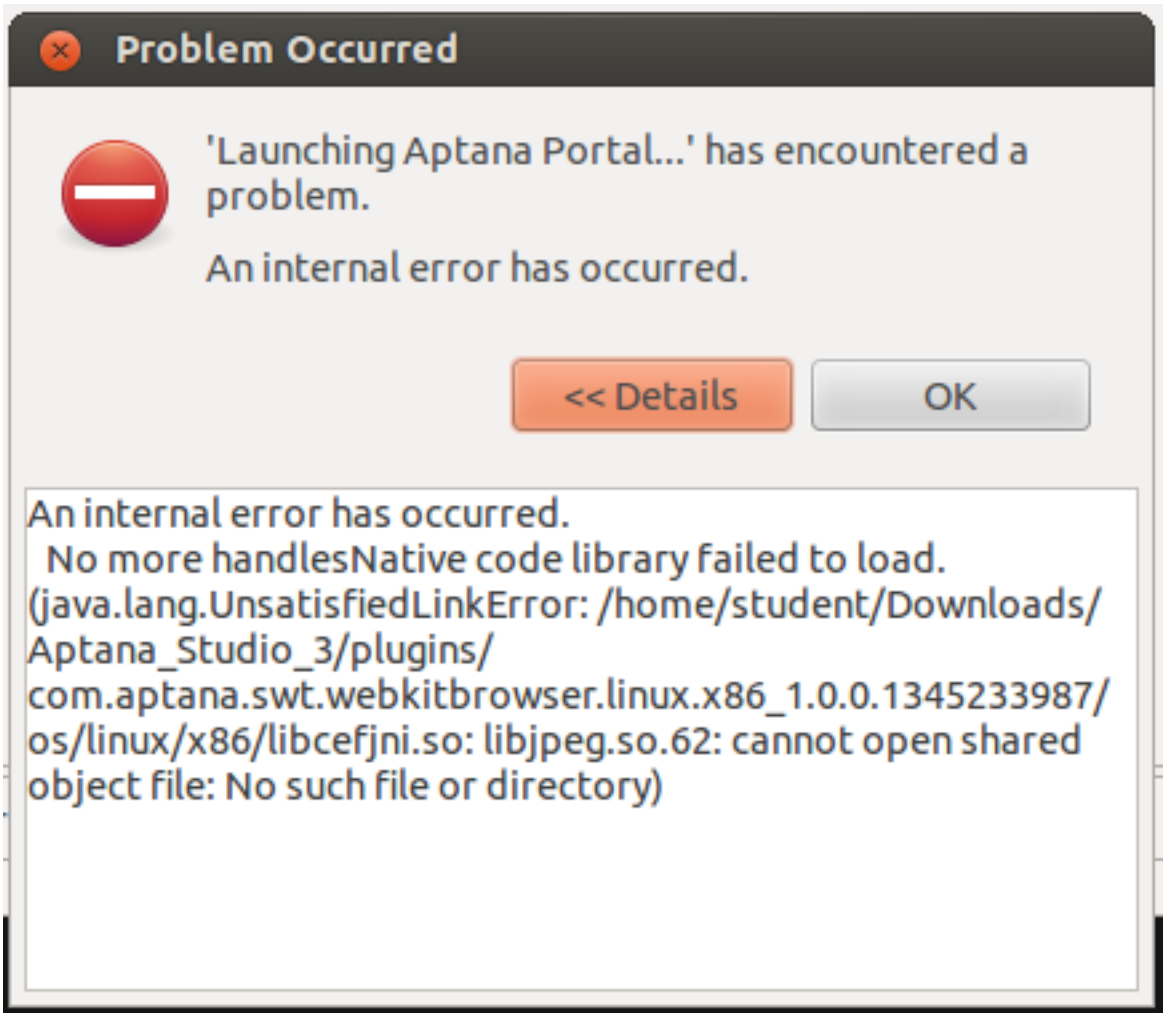


Instead use `~/workspace`, the standard Eclipse workspace path. You can tell Aptana to remember this workspace if you like.

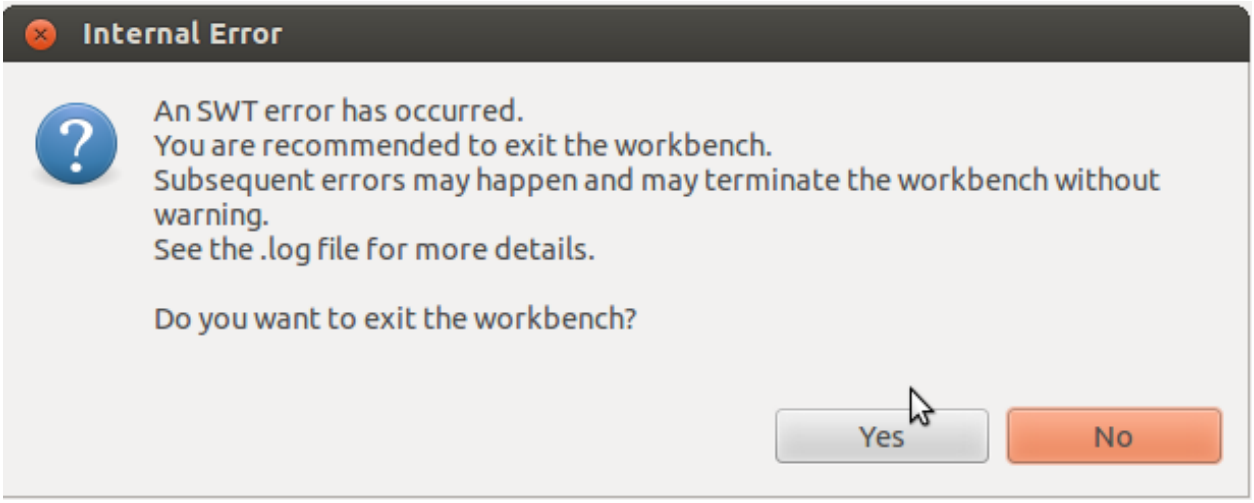


Harmless `libjpeg` Error

The first time you start Apatana Studio, you will get a frightening-looking error message, complaining that libjpeg.62.so is missing. This error is actually quite harmless - it is caused by Aptana trying to display it's one-time splash screen after a new install. To display the splash screen requires a JPEG graphic handling library that we have not installed.



Click OK, and Eclipse will ask if you want to exit. Say yes, wait for Eclipse to exit, then launch it again. You will only see this error message the first time; thereafter, launch will be error-free.



This bug can be avoided entirely by installing `libjpeg62`:

```
$ sudo apt-get install libjpeg62
```

However this is not strictly necessary, as the bug does not damage anything, and appears only the first time a new Apatana installation is run.

Installing Eclipse Plugins

Each Eclipse plugin has an *Update Site* URL, from which it can be installed.

To install a plugin in Eclipse, choose `Install New Software...` from the `Help` menu. Click the `Add...` button to add a new plugin repository. Put the plugin's *Update Site* URL in the `Location:` field.

Once you have added the plugin repository, check the box of the plugin you want to install. Click `Next >`, then click thru until it is installed. Normally Eclipse will want to restart itself after a new plugin has been installed.

Vwrapper

Vrapper is an Eclipse plugin providing VI-keys support. Only install this plugin if you are *certain* you want it.

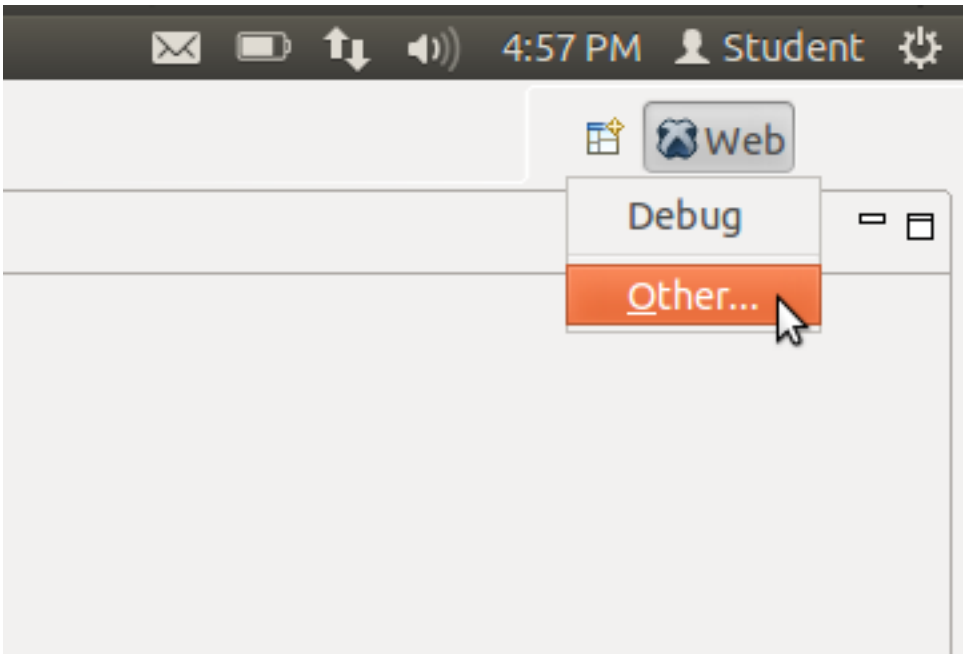
Update site:

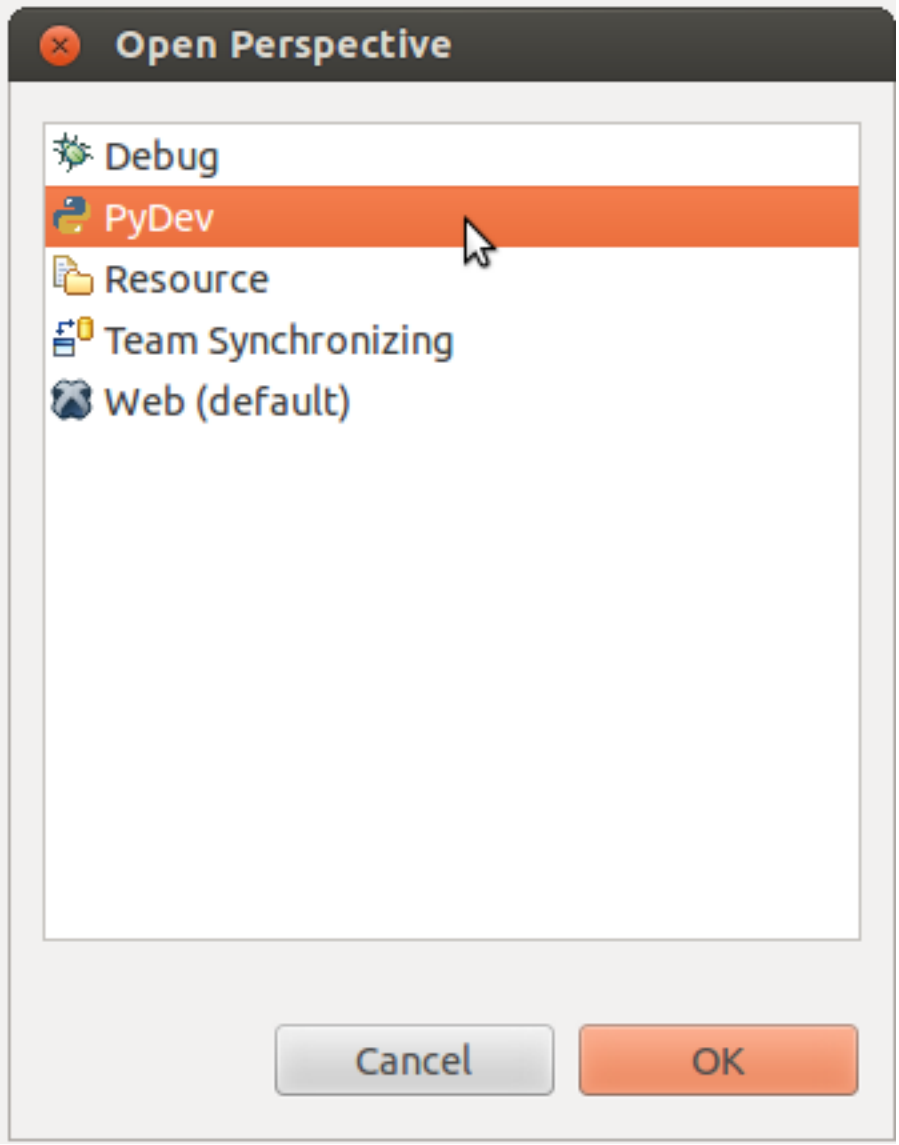
```
http://vrapper.sourceforge.net/update-site/stable
```

Python Perspective

Eclipse refers to collections of windows (*views* in Eclipse terminology) and their arrangement on screen as a *perspective*. There is a Python perspective available, pre-configured with the views one typically wants while working on Python code.

You can select the Python perspective by going to the *perspective menu* - it is in the upper right corner of the window, and looks like a grid with a plus sign. Click on it, select "Other...", and choose "PyDev Perspective" from the ensuing dialog.





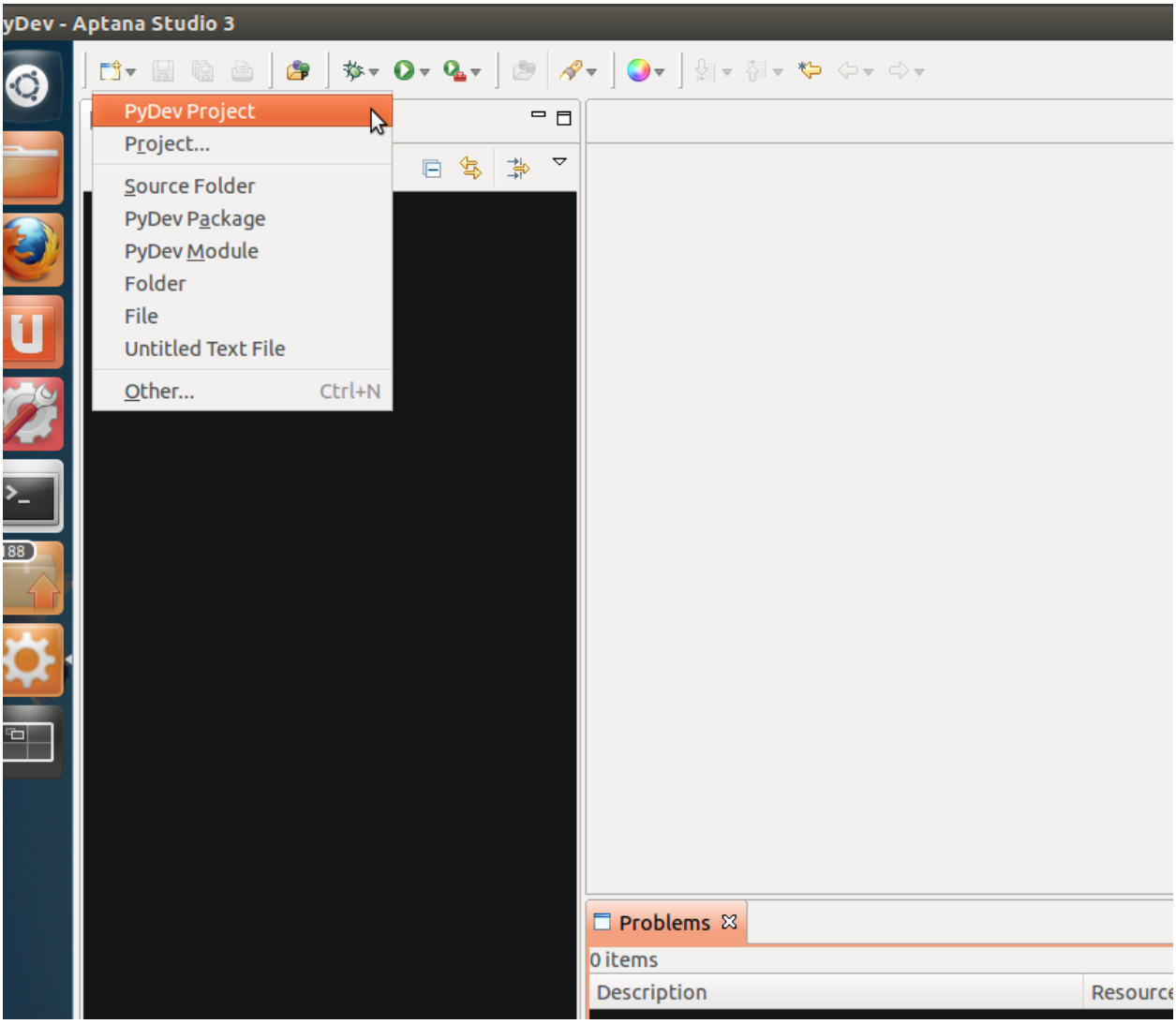
You will now be in the Python perspective.

Working with Virtual Environments

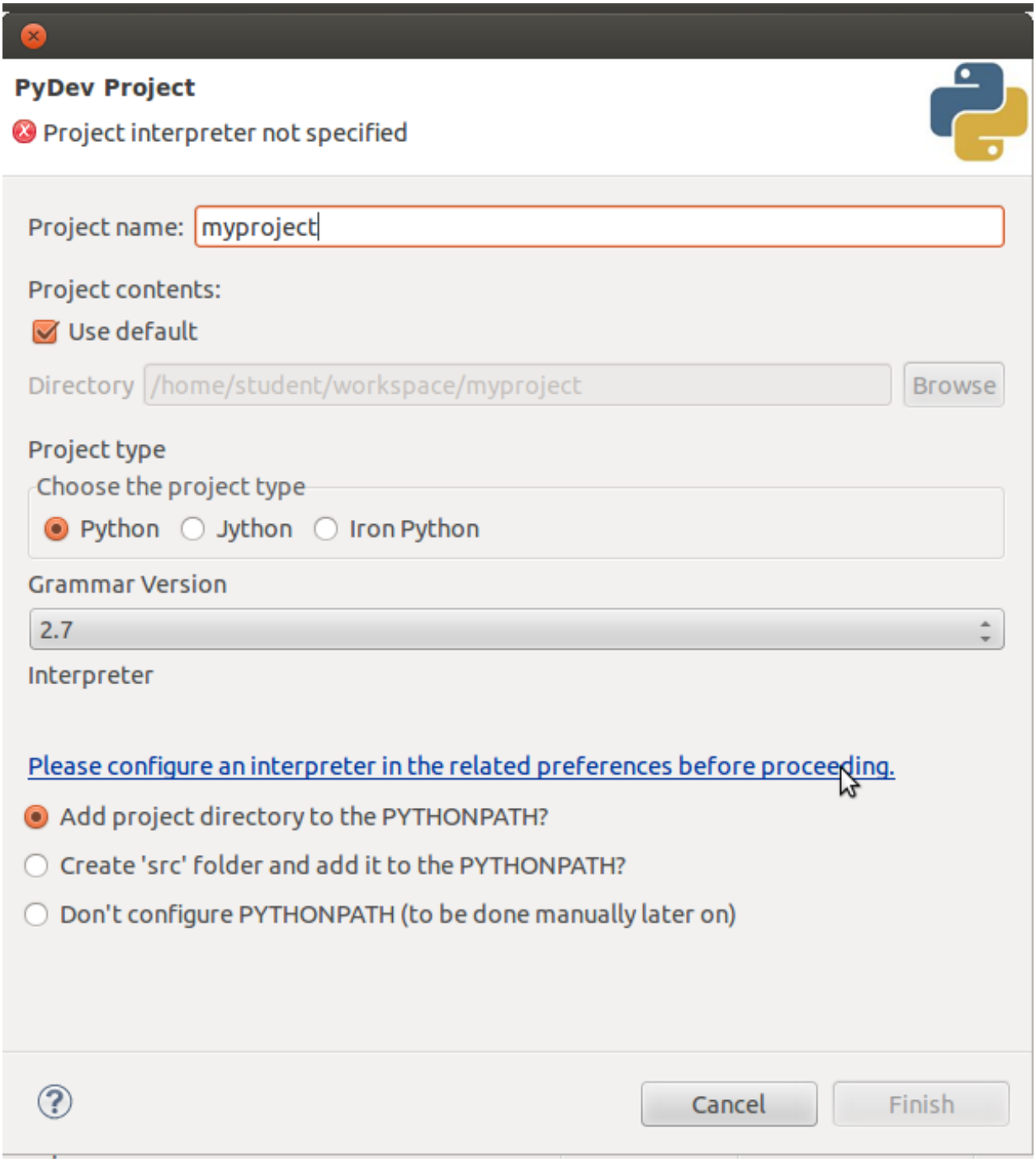
Unfortunately, Aptana is not aware of virtual environments by default. This can be worked around by manually configuring Aptana to use the Python interpreter from the virtual environment. We will configure the interpreter in the course of starting a new project below.

Starting a New Project

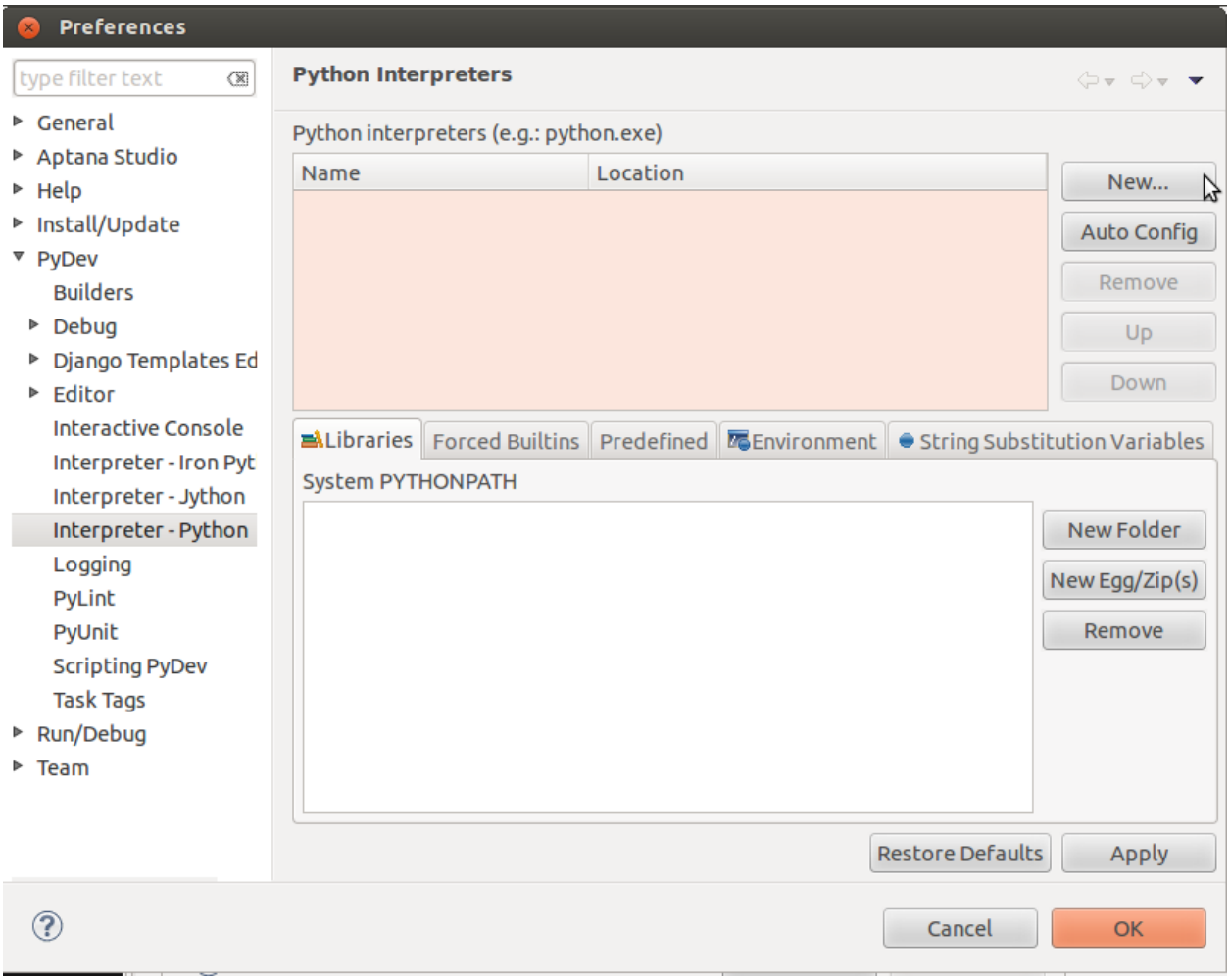
Start a new project by clicking on the new project menu - it looks like a window with a plus sign - in the upper left corner of the window. Click on the new project menu, and select "PyDev Project".



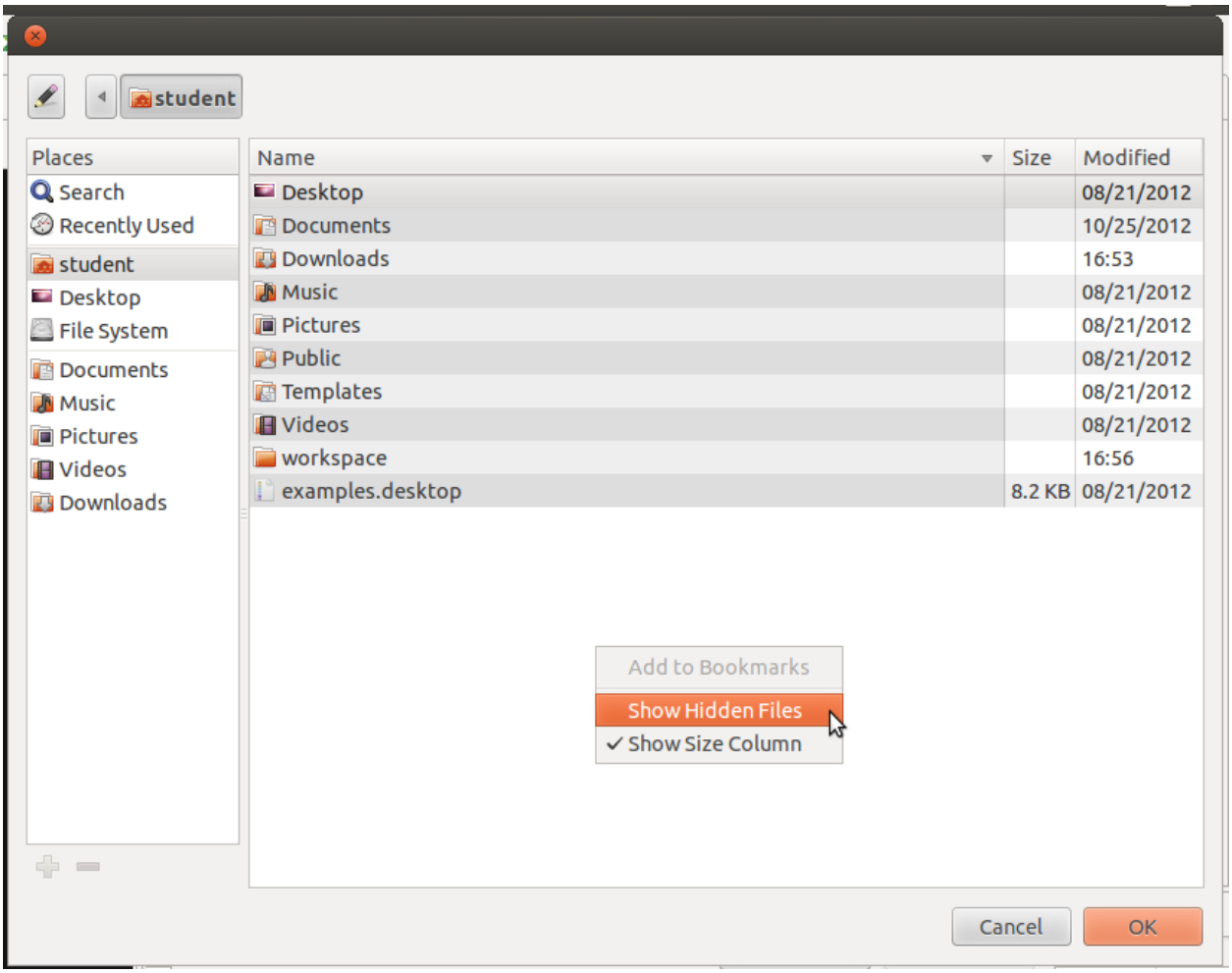
A new project dialog will appear. Fill in the name you want for your project, then click on the blue "Please configure an interpreter" link.



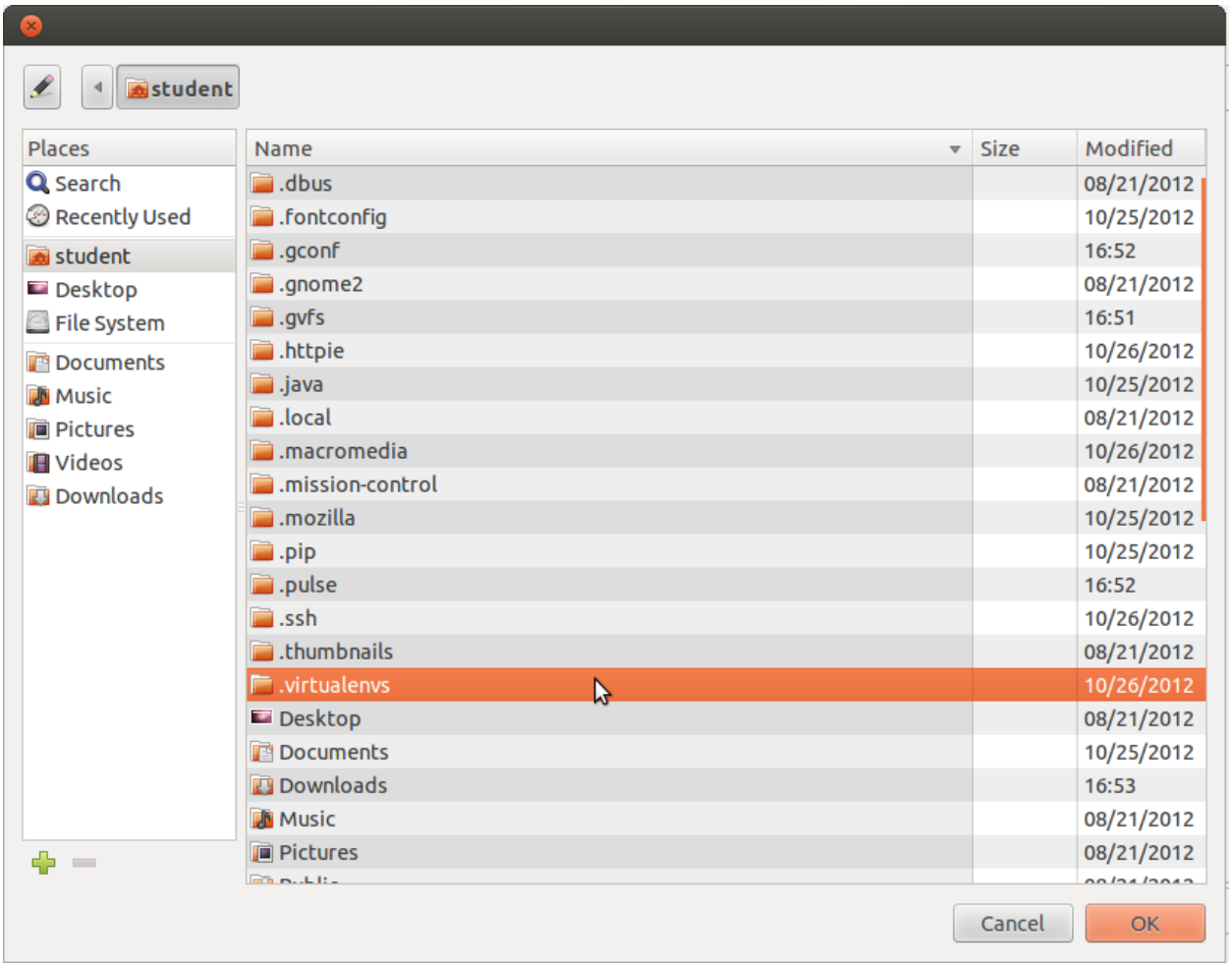
You will be taken to Eclipse's Python interpreter settings page.



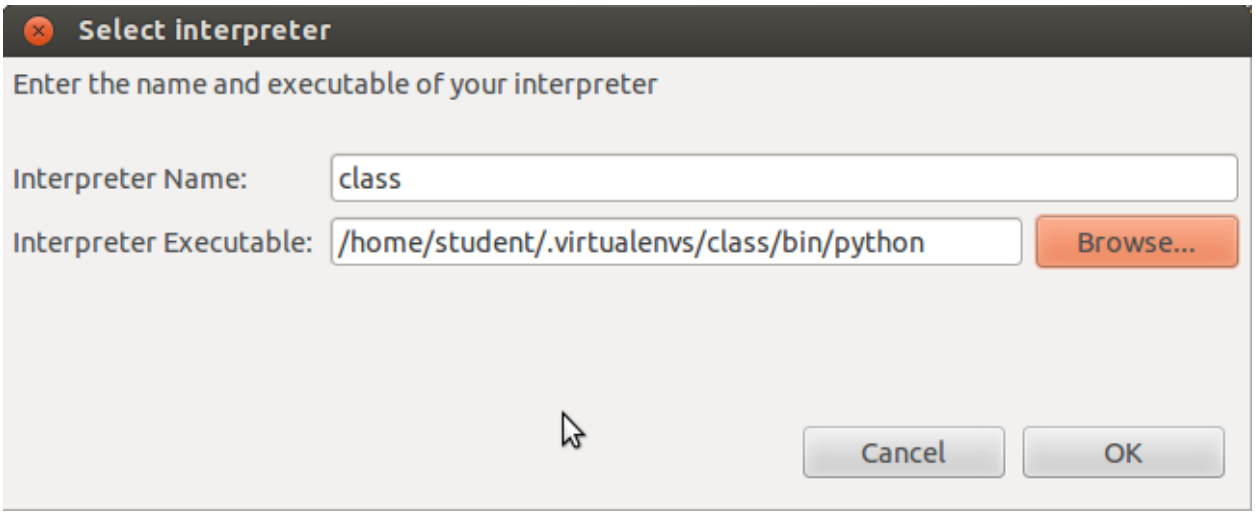
Click the "New..." button. A select file dialog will open; go to your home folder. Once there, right-click on the file list, and choose "Show Hidden Files" from the context menu.



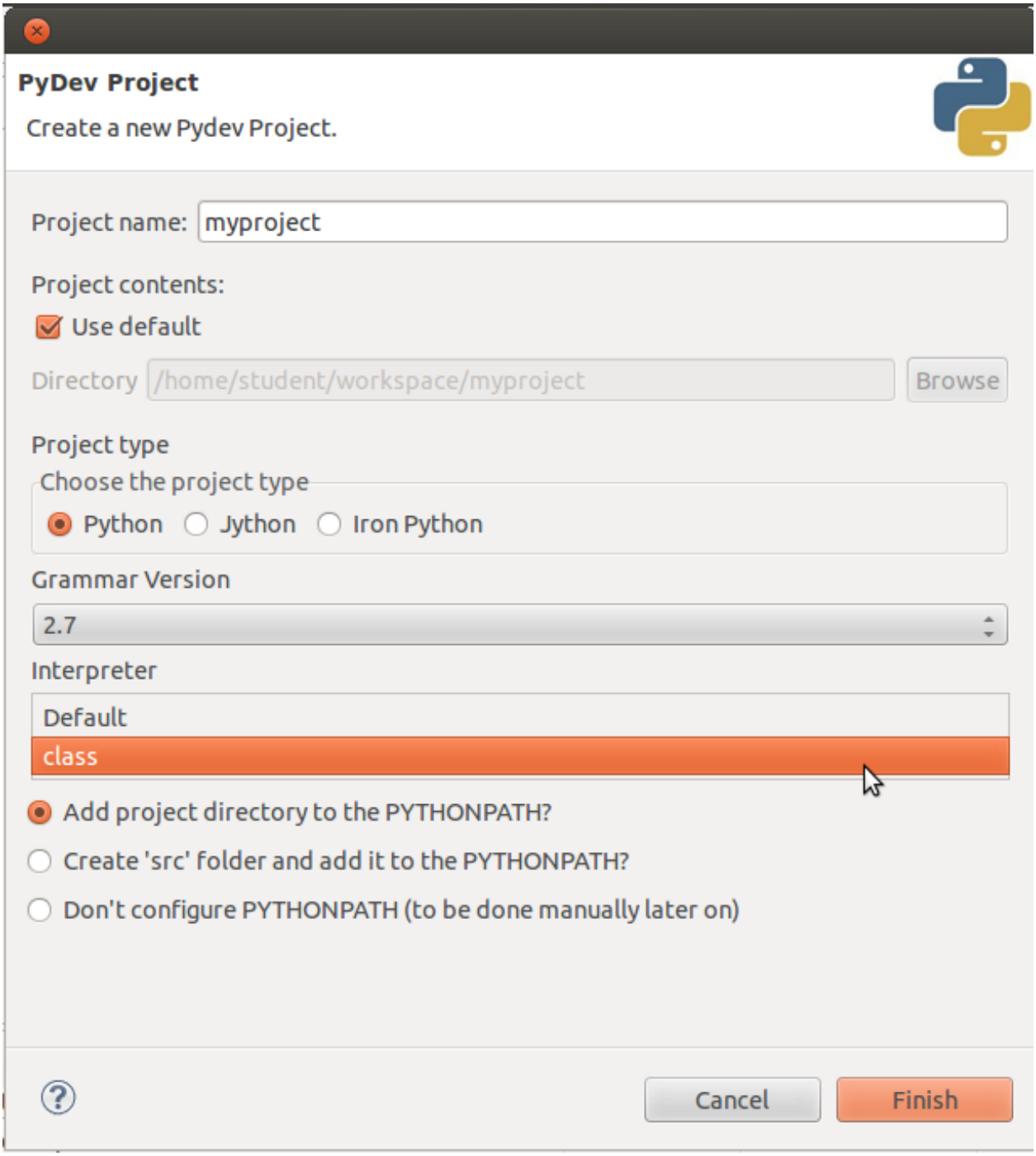
Click thru to select the `python` executable at `~/.virtualenvs/class/bin/python`.



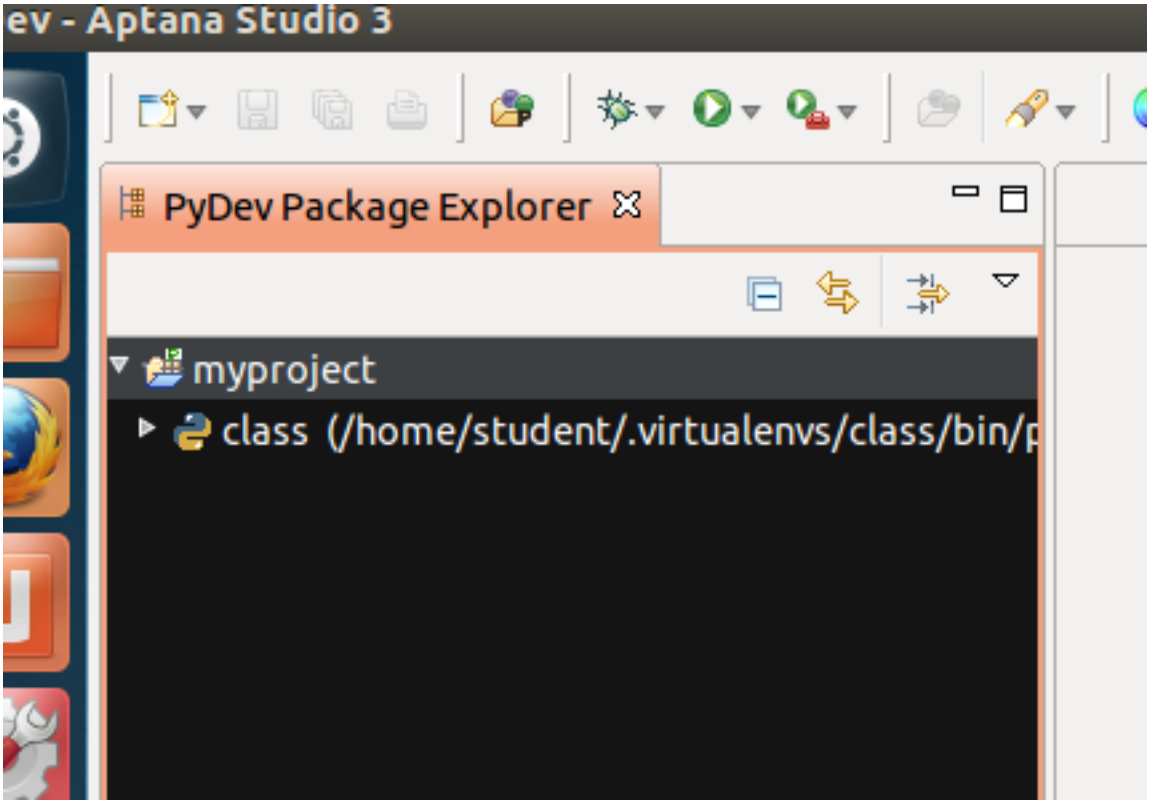
Click OK, and you will return to the Select Interpreter dialog, which will look like this:



Click “OK”, and you will return to the PyDev Project dialog. From the “Interpreter” popup menu, choose “class”, the virtualenv we just configured.



Click "Finish" and you will have successfully created a new project! Your new project will now be visible in the Project Explorer view.



Simulating Remote Hosts

Some of the class examples involve communicating with remote unix servers. Since the class will not necessarily have access to real remote servers, we will use `/etc/hosts` to simulate remote servers using only our local system.

Earlier we installed package `openssh-server`, the OpenSSH secure shell server. This will allow us to login to our VMs using SSH.

`/etc/hosts`

When a unix system attempts to resolve a domain name into an IP address, it first looks in the file `/etc/hosts`. If an entry is found, the name is resolved to that address. Otherwise, the system then queries a DNS server.

We will add two entries, named `newyork` and `seattle` to our `/etc/hosts` file, pointing those names to 127.0.0.1, the loopback IP address. (I.e. pointing them back at our local host.)

Initially your `/etc/hosts` file should look something like this:

```
127.0.0.1   localhost
127.0.1.1   sbtrain-vbox

# The following lines are desirable for IPv6 capable hosts
::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

Modify it to look like this:

```
127.0.0.1   localhost
127.0.0.1   newyork
127.0.0.1   seattle
127.0.1.1   sbtrain-vbox

# The following lines are desirable for IPv6 capable hosts
::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

SSH Key Setup

We will now configure our VM's SSH keys, so we can login without typing our credentials.

Generate a Key Pair

Generate a new public/private SSH keypair:

```
student@sbtrain-vbox:~$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/student/.ssh/id_rsa):
Created directory '/home/student/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/student/.ssh/id_rsa.
Your public key has been saved in /home/student/.ssh/id_rsa.pub.
The key fingerprint is:
38:a2:64:4a:9b:25:17:36:67:4b:a0:4a:42:ae:0e:90 student@sbtrain-vbox
The key's randomart image is:
+--[ RSA 2048]----+
| . .             |
|o.. .            |
|E+ + +           |
|* . * ..         |
|+ooo..o S        |
|++*. . .         |
|.+.              |
|                 |
|                 |
+-----------------+
student@sbtrain-vbox:~/.ssh$ ls
id_rsa  id_rsa.pub
```

Authorized Keys

Create an `authorized_keys` file containing the newly created public key:

```
student@sbtrain-vbox:~/.ssh$ cat id_rsa.pub >> authorized_keys
```

Verify Key Fingerprints

For each of our simulated hosts, we will need to verify the SSH key fingerprint one time before we can do fully automated logins:

```
student@sbtrain-vbox:~/.ssh$ ssh seattle
The authenticity of host 'seattle (127.0.0.1)' can't be established.
ECDSA key fingerprint is f3:c7:4b:87:c2:31:6d:ef:44:45:85:9a:21:e6:3c:7b.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'seattle' (ECDSA) to the list of known hosts.
Welcome to Ubuntu 12.04.1 LTS (GNU/Linux 3.2.0-29-generic-pae i686)

 * Documentation:  https://help.ubuntu.com/


The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

student@sbtrain-vbox:~$ exit
logout
Connection to seattle closed.
student@sbtrain-vbox:~/.ssh$
```

Repeat the same command for `newyork` and `localhost`.

Automatic Login

If you have completed all steps above successfully, you should now be able to login to any of our "remote" hosts without any keyboard interaction:

```
student@sbtrain-vbox:~/.ssh$ ssh seattle
Welcome to Ubuntu 12.04.1 LTS (GNU/Linux 3.2.0-29-generic-pae i686)

 * Documentation:  https://help.ubuntu.com/

Last login: Mon Nov  5 16:50:03 2012 from localhost
student@sbtrain-vbox:~$ exit
logout
Connection to seattle closed.
student@sbtrain-vbox:~/.ssh$
```

Lab Files

Code templates are provided as a starting point for several labs. Download the labs tarball.

You can extract the files using the `tar` command:

```
student@sbtrain-vbox:~/workspace$ tar zxvf /path/to/labs.tgz
labs/
labs/fabric/
labs/fabric/uptime.py
```

```
labs/rest_api/
labs/rest_api/list_repos.py
student@sbtrain-vbox:~/workspace$
```

CHAPTER 3

Using Fabric for SSH

Fabric is a library and command-line tool for streamlining the use of SSH for application deployment or systems administration tasks.

It provides a basic suite of operations for executing local or remote shell commands (normally or via sudo) and uploading/downloading files, as well as auxiliary functionality such as prompting the running user for input, or aborting execution.

`fab` Command

Fabric provides a command line utility `fab`, which reads its configuration from a file named `fabfile.py` in directory from which it is run. A typical fabfile contains one or more functions to be executed on a group of remote hosts.

The following example provides functions to check free disk space and host type, as well as defining a group of hosts on which to run:

```
from fabric.api import run

def host_type():
    run('uname -s')

def diskspace():
    run('df')
```

Once a task is defined, it may be run on one or more servers, like so:

```
(sysadmin)$ fab -H newyork,seattle host_type
[newyork] run: uname -s
[newyork] out: Linux
[seattle] run: uname -s
[seattle] out: Linux

Done.
```

```
Disconnecting from newyork... done.
Disconnecting from seattle... done.
```

Rather than supplying hostnames on the command line, you can also define the group of hosts on which tasks will be run with `env.hosts`:

```
from fabric.api import run
from fabric.api import env

env.hosts = [
    'newyork',
    'seattle',
    'localhost',
    ]

def host_type():
    run('uname -s')

def diskspace():
    run('df')
```

Now run `fab` without a `-H` argument:

```
(sysadmin)$ fab host_type
[newyork] run: uname -s
[newyork] out: Linux
[seattle] run: uname -s
[seattle] out: Linux
[localhost] run: uname -s
[localhost] out: Linux

Done.
Disconnecting from newyork... done.
Disconnecting from seattle... done.
Disconnecting from localhost... done.
```

Task arguments

It's often useful to pass runtime parameters into your tasks, just as you might during regular Python programming. Fabric has basic support for this using a shell-compatible notation: `<task name>:<arg>,<kwarg>=<value>,...` It's contrived, but let's extend the above example to say hello to you personally:[1]

```
def hello(name="world"):
   print("Hello %s!" % name)
```

By default, calling `fab hello` will still behave as it did before; but now we can personalize it:

```
(sysadmin)$ fab hello:name=Jeff
Hello Jeff!

Done.
```

Those already used to programming in Python might have guessed that this invocation behaves exactly the same way:

[1] http://docs.fabfile.org/en/1.4.2/tutorial.html#task-arguments

```
(sysadmin)$ fab hello:Jeff
Hello Jeff!

Done.
```

For the time being, your argument values will always show up in Python as strings and may require a bit of string manipulation for complex types such as lists. Future versions may add a typecasting system to make this easier.

Library Usage

In addition to use via the fab tool, Fabric's components may be imported into other Python code, providing a Pythonic interface to the SSH protocol suite at a higher level than that provided by e.g. the ssh library (which Fabric itself uses.)[2]

Lab – Average Uptime

Consider the case where we want to collect average uptime from a group of hosts. File `labs/fabric/uptime.py` will get you started:

```
'''
Lab - Average Uptime

Write a script that uses the Fabric library to poll a group of hosts for their
uptimes, and displays their average uptime for the user.
'''


from fabric import tasks
from fabric.api import run
from fabric.api import env
from fabric.network import disconnect_all


env.hosts = [
    'newyork',
    'seattle',
    'localhost',
    ]

def uptime():
    res = run('cat /proc/uptime')
    #
    # Now, examine the result and extract the average uptime
    #


def main():
    uts_dict = tasks.execute(uptime)
    #
    # Now, calculate average uptime...
    #
    disconnect_all() # Call this when you are done, or get an ugly exception!
```

[2] http://stackoverflow.com/a/8166050

```
34
35
36 if __name__ == '__main__':
37     main()
38
```

CHAPTER 4

Accessing RESTful APIs

What is a RESTful API?

A 'RESTful API' is a remote API that follows the *REST* style of software architecture.

REST - Representational State Transfer

REpresentational State Transfer (REST) is a style of software architecture for distributed systems such as the World Wide Web. REST has emerged as a predominant Web service design model.[1]

The term representational state transfer was introduced and defined in 2000 by Roy Fielding in his doctoral dissertation. Fielding is one of the principal authors of the Hypertext Transfer Protocol (HTTP) specification versions 1.0 and 1.1.

Conforming to the **REST constraints** is generally referred to as being "RESTful".

The REST Constraints

The REST architectural style describes the following six constraints applied to the architecture, while leaving the implementation of the individual components free to design:[2]

Client–server

A uniform interface separates clients from servers. This separation of concerns means that, for example, clients are not concerned with data storage, which remains internal to each server, so that the portability of client code is improved. Servers are not concerned with the user interface or user state, so that servers can be simpler and more scalable. Servers and clients may also be replaced and developed independently, as long as the interface between them is not altered.

Stateless

The client–server communication is further constrained by no client context being stored on the server between requests. Each request from any client contains all of the information necessary to service the request, and any session state is held in the client.

[1] http://en.wikipedia.org/wiki/Representational_state_transfer
[2] http://en.wikipedia.org/wiki/Representational_state_transfer#Constraints

Cacheable

As on the World Wide Web, clients can cache responses. Responses must therefore, implicitly or explicitly, define themselves as cacheable, or not, to prevent clients reusing stale or inappropriate data in response to further requests. Well^managed caching partially or completely eliminates some client–server interactions, further improving scalability and performance.

Layered system

A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way. Intermediary servers may improve system scalability by enabling load^balancing and by providing shared caches. They may also enforce security policies.

Code on demand (optional)

Servers are able temporarily to extend or customize the functionality of a client by the transfer of executable code. Examples of this may include compiled components such as Java applets and client^side scripts such as JavaScript.

Uniform interface

The uniform interface between clients and servers, discussed below, simplifies and decouples the architecture, which enables each part to evolve independently. The four guiding principles of this interface are detailed below.

The only optional constraint of REST architecture is code on demand. If a service violates any other constraint, it cannot strictly be considered RESTful.

Complying with these constraints, and thus conforming to the REST architectural style enables any kind of distributed hypermedia system to have desirable emergent properties, such as performance, scalability, simplicity, modifiability, visibility, portability, and reliability.

JSON - Javascript Object Notation

JSON, or *JavaScript Object Notation*, is a text-based open standard designed for human-readable data interchange. It is derived from the JavaScript scripting language for representing simple data structures and associative arrays, called objects. Despite its relationship to JavaScript, it is language-independent, with parsers available for many languages.[3]

The JSON format was originally specified by Douglas Crockford, and is described in RFC 4627. The official Internet media type for JSON is `application/json`. The JSON filename extension is `.json`.

The JSON format is often used for serializing and transmitting structured data over a network connection. It is used primarily to transmit data between a server and web application, serving as an alternative to XML.

Citations

Requests Library

Requests is an ISC Licensed HTTP library, written in Python, for human beings.[1]

Requests takes all of the work out of Python HTTP/1.1 — making your integration with web services seamless. There's no need to manually add query strings to your URLs, or to form-encode your POST data. Keep-alive and HTTP connection pooling are 100% automatic, powered by urllib3, which is embedded within Requests.

[3] http://en.wikipedia.org/wiki/JSON
[1] http://docs.python-requests.org/

Generate a Github OAuth2 Token

There are two ways to authenticate with the GitHub API: HTTP basic auth, and OAuth2.[1] It is preferable to use OAuth2, so your script can run without user input, and without storing your password.

The OAauth2 token can be sent in the request header, or as a parameter. We will send it as a header in later examples.

POST Request

First, we will prompt the user for username/password, and compose a POST request to the API. The request format is documented in the OAuth section of the Github API docs.

```
GITHUB_API = 'https://api.github.com'


import requests
import json
from urlparse import urljoin


def main():
    #
    # User Input
    #
    username = raw_input('Github username: ')
    password = raw_input('Github password: ')
    #
    # Compose Request
    #
    url = urljoin(GITHUB_API, 'authorizations')
    payload = {}
    res = requests.post(
        url,
        auth = (username, password),
        data = json.dumps(payload),
        )
    print res.text

if __name__ == '__main__':
    main()
```

Let's give it a try:

```
(class)$ python authtoken.py
Github username: jmcvetta
Github password: fooba
```

That's not good - our password is shown when we type it!

Password Privacy

We can protect the user's privacy while inputting their password with the `getpass` library. While we're at it, we can prompt the user for an optional note to describe how this token will be used.

[1] http://developer.github.com/v3/#authentication

```
GITHUB_API = 'https://api.github.com'


import requests
import getpass
import json
from urlparse import urljoin


def main():
    #
    # User Input
    #
    username = raw_input('Github username: ')
    password = getpass.getpass('Github password: ')
    note = raw_input('Note (optional): ')
    #
    # Compose Request
    #
    url = urljoin(GITHUB_API, 'authorizations')
    payload = {}
    if note:
        payload['note'] = note
    res = requests.post(
        url,
        auth = (username, password),
        data = json.dumps(payload),
        )
    print res.text

if __name__ == '__main__':
    main()
```

Let's give it a try:

```
(class)$ python authtoken.py
Github username: jmcvetta
Github password:
Note (optional): admin script
{"scopes":[],"note_url":null,"created_at":"2012-10-21T05:32:30Z","url":"https://api.
↪github.com/authorizations/744660","app":{"url":"http://developer.github.com/v3/
↪oauth/#oauth-authorizations-api","name":"admin script (API)"},"updated_at":"2012-10-
↪21T05:32:30Z","token":"a977026974077e83e593744aa9308422e92a26bd","note":"admin␣
↪script","id":744660}
```

Seems to have worked! The response is a big JSON blob.

JSON Parsing

We can parse the JSON response and provide just the token, in nice human-readable form, to the user.

Explore the response data by setting a breakpoint and running our program in the debugger. Start by parsing `res.text` into JSON, and examining the keys.

The token lives in the creatively-named field `token`. We will extract it and print it for the user.

```
GITHUB_API = 'https://api.github.com'


import requests
import getpass
import json
from urlparse import urljoin


def main():
    #
    # User Input
    #
    username = raw_input('Github username: ')
    password = getpass.getpass('Github password: ')
    note = raw_input('Note (optional): ')
    #
    # Compose Request
    #
    url = urljoin(GITHUB_API, 'authorizations')
    payload = {}
    if note:
        payload['note'] = note
    res = requests.post(
        url,
        auth = (username, password),
        data = json.dumps(payload),
        )
    #
    # Parse Response
    #
    j = json.loads(res.text)
    token = j['token']
    print 'New token: %s' % token

if __name__ == '__main__':
    main()
```

Let's give it a try:

```
(class)$ python authtoken.py
Github username: jmcvetta
Github password:
Note (optional): admin script
New token: 9c0e2ab295ee0e92130142ad3c90bbf5fe93642f
```

Bingo - it worked!

Error Handling

But what if we don't type the right username/password combo?

```
(class)$ python authtoken.py
Github username: jmcvetta
Github password:
Note (optional):
Traceback (most recent call last):
```

```
  File "authtoken.2.py", line 46, in <module>
    main()
  File "authtoken.2.py", line 42, in main
    token = j['token']
KeyError: 'token'
```

Gross.

Once again we can run our program in the debugger to explore the server's response. It looks like we have a `res.status_code` of 401. Any HTTP response code 400 or above indicates an error. It also looks like the server helpfully provides a `message` field with an error message.

We can look for response codes >= 400 and present the user a friendly error message:

```
GITHUB_API = 'https://api.github.com'


import requests
import getpass
import json
from urlparse import urljoin


def main():
    #
    # User Input
    #
    username = raw_input('Github username: ')
    password = getpass.getpass('Github password: ')
    note = raw_input('Note (optional): ')
    #
    # Compose Request
    #
    url = urljoin(GITHUB_API, 'authorizations')
    payload = {}
    if note:
        payload['note'] = note
    res = requests.post(
        url,
        auth = (username, password),
        data = json.dumps(payload),
        )
    #
    # Parse Response
    #
    j = json.loads(res.text)
    if res.status_code >= 400:
        msg = j.get('message', 'UNDEFINED ERROR (no error description from server)')
        print 'ERROR: %s' % msg
        return
    token = j['token']
    print 'New token: %s' % token

if __name__ == '__main__':
    main()
```

Let's give it a try:

```
(class)$ python authtoken.py
Github username: jmcvetta
Github password:
Note (optional):
ERROR: Bad credentials
```

Now we have a friendly, useful program.

Lab – List Github Repositories

Let's suppose we want to list all the Github repositories associated with the authenticated user.

Github defines several types of repository that can be listed:

- `all`
- `owner`
- `public`
- `private`
- `member`

Our application should query for `all` by default, but allow the user to specify a different type.

Argparse

A production quality command line utility will typically require the ability to accept arguments when it is called. While it is *possible* to parse these arguments manually from `sys.argv`, the recommended technique is to use the `argparse` library. Argparse is part of the Python standard library, and is very well documented.

Note: Argparse was introduced in Python 2.7. Prior versions of Python include an earlier library, `optparse`. Argparse is intended as a full replacement for optparse, and so usage of the latter is deprecated.

Lab

File `labs/rest_api/list_repos.py` will get you started:

```
'''
Lab - List Repositories

Write a script that queries the Github API for repositories
belonging to the authenticated user.

For each repo, print out its name, its description, and the number
of open issues.
'''

API_TOKEN = 'YOUR_TOKEN_GOES_HERE'
VALID_TYPES = ['all', 'owner', 'public', 'private', 'member']

import requests
import argparse

def main():
```

```
    parser = argparse.ArgumentParser(description='List Github repositories.')
    parser.add_argument('-t', '--type',
        nargs = 1,
        dest = 'type',
        default = 'all',
        metavar = 'TYPE',
        choices = VALID_TYPES,
        help = 'What type of repos to list',
        )
    args = parser.parse_args() # You can access the 'type' argument as args.type
    #
    # Use the authentication token we generated in the previous example
    #
    headers = {
        'Authorization': 'token %s' % API_TOKEN
        }
    #
    # Now, build a REST request, and parse the server's response...
    #


if __name__ == '__main__':
    main()
```

CHAPTER 5

Django

What is Django?

Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design.[1]

Django focuses on automating as much as possible and adhering to the DRY principle.

Django Tutorial

We will follow a modified version of the standard Django Tutorial.

Todo

Include text of a modified (to include class based views, etc) Django tutorial.

[1] http://djangoproject.com

CHAPTER 6

Work In Progress

This curriculum is a work in progress. Many sections are missing or incomplete. There may still be some TODOs:

Todo

Include text of a modified (to include class based views, etc) Django tutorial.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/advanced-python/checkouts/latest/django/tutorial.rst, line 10.)