Pants Documentation

Release 1.0.1

Christopher Davis

Jul 08, 2017

Contents

Pants is a lightweight framework for writing asynchronous network applications in Python. Pants is simple, fast and elegant.

Pants is available under the Apache License, Version 2.0

An incomplete feature list:

- Single-threaded, asynchronous, callback-oriented.
- TCP networking - clients and servers!
- IPv4, IPv6 and UNIX socket families.
- SSL/TLS support for all that security stuff.
- Basic scheduling and timers.
- A speedy HTTP server with a handy WebSockets implementation.
- A simple web framework and support for WSGI.

And it's all so, so easy to use. Check it out:

```
from pants import Engine, Server, Stream

class Echo(Stream):
    def on_read(self, data):
        self.write(data)

Server(Echo).listen(4040)
Engine.instance().start()
```

Here's a web example for good measure:

```
from pants.web import Application

app = Application()

@app.route('/')
def hello(request):
    return "Hello, World!"

app.run()
```

And here's how you get Pants:

```
pip install pants
```

Want to get started? There's plenty to do:

- Fork ecdavis/pants on GitHub.
- Join the IRC channel, #pantspowered on Freenode.
- Read this documentation!

CHAPTER 1

Documentation

User Guide

Pants is a network programming framework for Python. It is simple, fast and elegant. Pants provides the programmer with the basic tools they need to write responsive, high-throughput and highly-concurrent network applications. At its core, Pants consists of three things:

- **Engines**: efficient, asynchronous event loops.
- **Channels**: non-blocking wrappers around socket objects.
- **Timers**: non-blocking helpers for delayed execution of code.

Overview

All Pants applications share a similar architecture. Channels and timers are added to an engine. The engine runs an event loop and manages timer scheduling. As events are raised on sockets, the engine dispatches those events (read, write, close, etc.) to the relevant channel to be handled by user code. Timers are executed and possibly rescheduled by the engine as they expire. Writing a Pants application consists of defining your event-handling logic on custom channel classes, scheduling timers to be executed and starting the event loop. Pants makes writing efficient network applications simple through the use of elegant abstractions.

Pants is an asynchronous, callback-oriented framework. Being asynchronous, it is important that your code does not block the main process. Blocking code prevents Pants from efficiently polling for socket events, and has a significant, negative effect on performance. To eliminate the need for blocking code, Pants uses a callback-oriented design. Blocking operations like reading and writing data to a socket are performed in the background. When these operations complete, callback methods are invoked to notify user code. To get a better example of how Pants applications work, take a look at a few examples or read through the *tutorial*.

Getting Started

Getting started with Pants is easy. Pants can be installed either from the Python Package Index or from source. Pants requires Python version 2.6 or 2.7.

You can install Pants using your favourite Python package manager:

```
pip install pants
```

Or from source:

```
wget https://github.com/ecdavis/pants/tarball/pants-1.0.0-beta.3
tar xvfz pants-1.0.0-beta.3.tar.gz
cd pants-1.0.0-beta.3
python setup.py install
```

Using the development version

Using the development version of Pants will give you access to the latest features as well as the latest bugs. If you're interested in contributing code to Pants, this is the version you should work with. Otherwise, it's suggested that you stick to a release version. You can clone the repository like so:

```
git clone git://github.com/ecdavis/pants
```

Many people also find it useful to add their repository directory to Python's path, or to create a symbolic link from the repository directory to Python's site-packages directory to allow them to import Pants in any Python script.

Tutorial

What follows is a simple tutorial designed to introduce you to the core parts of Pants' API and demonstrate how to write simple Pants applications. This tutorial is by no means an exhaustive tour of Pants' many features, but should serve as an excellent starting point for someone new to the framework or to asynchronous network programming in general.

Writing a simple server

We're going to begin by writing an echo server. This is like the "Hello, World!" of networking frameworks, but it's nonetheless a good place to start. Create a file containing the following code:

```
from pants import Engine, Server, Stream

class Echo(Stream):
    def on_read(self, data):
        self.write(data)

server = Server(ConnectionClass=Echo)
server.listen(4040)
Engine.instance().start()
```

Now run it and, in another terminal, connect to the server using telnet:

```
telnet localhost 4040
```

Try entering some data and you'll find that it gets echoed right back to you. To get a better idea of what's happening in this application, we'll run through the code line by line:

```
class Echo(Stream):
    def on_read(self, data):
        self.write(data)
```

We begin by defining a class, `Echo`, which subclasses Pants' *`Stream`* class. Instances of *`Stream`* and its subclasses are what Pants calls 'channels.' They represent connections from the local host to a remote host or vice-versa. Channels are basically just wrappers around `socket` objects that deal with all the nitty-gritty, low-level stuff so that you don't have to. You implement most of your application's logic by defining callback methods on your channel classes. `on_read` is one such method. As the name suggests, `on_read` will get called any time data is read from the channel. The incoming data is passed to the callback for use by your application. In this case, we've chosen to immediately write it back to the channel, thereby implementing the echo protocol.

Having defined our application logic, we now need to get the server up and running:

```
server = Server(ConnectionClass=Echo)
server.listen(4040)
Engine.instance().start()
```

We create a new instance of Pants' *`Server`* class and pass it our *`Stream`* subclass, `Echo`. *`Server`* instances are channels which represent sockets that are listening for new connections to the local host. When a new connection is made, the *`Server`* will automatically wrap that connection with an instance of its `ConnectionClass`. In this case, new connections will be wrapped with instances of our `Echo` class.

After creating the server, we tell it to listen for new connections on port 4040 and then we start the global engine. All Pants applications have an engine at their core - it's responsible for running a powerful event loop that listens for new events on sockets and dispatches those events to the appropriate channels.

We've only written 7 lines of code, but we've already covered a great deal of Pants' core functionality. Before moving on, try messing around with the code a little bit and see what happens:

- Delete the `ConnectionClass` parameter in the *`Server`* constructor.
- Comment out the *`listen()`* call.
- Comment out the *`start()`* call.

Kicking it up a notch

Now that we've covered the basics we can move on to something a little more interesting.

```
from pants import Engine, Server, Stream

class BlockEcho(Stream):
    def on_connect(self):
        self.read_delimiter = 8

    def on_read(self, block):
        self.write(block + '\r\n')

server = Server(ConnectionClass=BlockEcho)
server.listen(4040)
Engine.instance().start()
```

The `on_read` method is basically the same as before, we've just added a newline to the end of the data before writing it. We've also added a new event handler method: `on_connect`. As the name suggests, this gets called when the channel's connection is first established. In `on_connect` we set the value of the channel's *`read_delimiter`* attribute, and this is where things get neat. The *`read_delimiter`* changes the way Pants passes data to `on_read`. Instead of being passed on as soon as it arrives, data is internally buffered and passed to `on_read` in blocks of 8 bytes. See what happens when you run this application and connect to it as you did before. It's a simple idea, but the *`read_delimiter`* is one of Pants' most powerful features.

The *`read_delimiter`* isn't limited to being a number of bytes, either. Here are some experiments for you to try:

- Set the *read_delimiter* to a short string.
- Set the *read_delimiter* to a compiled regex object.
- Set the *read_delimiter* to None.

Taking it to another level

Up until now we've been using Pants' regular *Server* class and it's suited our needs perfectly. There are times, however, where you may need to define custom behaviour on your server channels. This is achieved by subclassing *Server*:

```
class EchoLineToAllServer(Server):
    ConnectionClass = EchoLineToAll

    def write_to_all(self, data):
        for channel in self.channels.itervalues():
            if channel.connected:
                channel.write(data)
```

All very straight-forward. We defined a new method on the server that writes data to all connected channels. We also overrode the default `ConnectionClass` attribute, meaning that we'll no longer need to pass in our connection class to the constructor. Starting the server now looks like this:

```
EchoLineToAllServer().listen(4040)
Engine.instance().start()
```

For the sake of completeness, here's the `EchoLineToAll` connection class used by the above server:

```
class EchoLineToAll(Stream):
    def on_connect(self):
        self.read_delimiter = '\r\n'
        self.server.write_to_all("Connected: %s\r\n" % self.remote_address[0])

    def on_read(self, line):
        self.server.write_to_all("%s: %s\r\n" % (self.remote_address[0], line))

    def on_close(self):
        self.server.write_to_all("Disconnected: %s\r\n" % self.remote_address[0])
```

As you can see, channels retain a reference to the server that they belong to. In this case, we're also using the *remote_address* property as a channel-specific identifier.

That's it for the basic tutorial, but there's plenty more you can do here:

- Experiment with different *read_delimiter* values to change the way connections process data. You might try implementing a packet-oriented protocol.
- Write a client for your server using Pants. You basically know how already, just take a look at *connect()* and you'll be good to go.
- We can't have people communicating through unencrypted channels like this. Secure your chat server using Pants' SSL support. Take a look at *startSSL()* to get started.

Core

pants

The core Pants classes and objects.

Exports the global engine, *pants.stream.Stream* and *pants.server.Server*.

pants.engine

Asynchronous event processing and timer scheduling.

Pants applications are powered by instances of the *Engine* class. An *Engine* instance keeps track of active channels, continuously checks them for new events and raises those events on the channel when they occur. The *Engine* class also provides the timer functionality which allows callable objects to be invoked after some delay without blocking the process.

Engines

Pants' engines are very simple to use. After you have finished initializing your application, simply call *start()* to enter the blocking event loop. *stop()* may be called at any time to cause a graceful exit from the event loop. If your application has a pre-existing event loop you can call the *poll()* method on each iteration of that loop rather than using *start()* and *stop()*. Ideally, *poll()* should be called many times each second to ensure that events are processed efficiently and timers are executed on schedule.

The global engine instance is returned by the *instance()* classmethod. It is not required that you use the global engine instance, but it is strongly recommended. By default, new channels are automatically added to the global engine when they are created. Channels can be added to a specific engine by passing the engine instance as a keyword argument to the channel's constructor. If a *Server* is added to a non-default engine, any connections it accepts will also be added to that engine.

Timers

In addition to managing channels, Pants' engines can also schedule timers. Timers are callable objects that get invoked at some point in the future. Pants has four types of timers: callbacks, loops, deferreds and cycles. Callbacks and loops are executed each time *poll()* is called - callbacks are executed once while loops are executed repeatedly. Deferreds and cycles are executed after a delay specified in seconds - deferreds are executed once while cycles are executed repeatedly.

Engine has methods for creating each of the four types of timers: *callback()*, *loop()*, *defer()* and *cycle()*. Each of these methods is passed a callable to execute as well as any number of positional and keyword arguments:

```
engine.callback(my_callable, 1, 2, foo='bar')
```

The timer methods all return a callable object which can be used to cancel the execution of the timer:

```
cancel_cycle = engine.cycle(10.0, my_callable)
cancel_cycle()
```

Any object references passed to a timer method will be retained in memory until the timer has finished executing or is cancelled. Be aware of this when writing code, as it may cause unexpected behaviors should you fail to take these references into account. Timers rely on their engine for scheduling and execution. For best results, you should either schedule timers while your engine is running or start your engine immediately after scheduling your timers.

Pollers

By default, Pants' engines support the `epoll`, `kqueue` and `select` polling methods. The most appropriate polling method is selected based on the platform on which Pants is running. Advanced users may wish to use a different polling method. This can be done by defining a custom poller class and passing an instance of it to the *`Engine`* constructor. Interested users should review the source code for an understanding of how these classes are defined and used.

Engine

class `pants.engine.`**Engine**(*poller=None*)

The asynchronous engine class.

An engine object is responsible for passing I/O events to active channels and running timers asynchronously. Depending on OS support, the engine will use either the `epoll()`, `kqueue()` or `select()` system call to detect events on active channels. It is possible to force the engine to use a particular polling method, but this is not recommended.

Most applications will use the global engine object, which can be accessed using *`instance()`*, however it is also possible to create and use multiple instances of *`Engine`* in your application.

An engine can either provide the main loop for your application (see *`start()`* and *`stop()`*), or its functionality can be integrated into a pre-existing main loop (see *`poll()`*).

Argument	Description
poller	*Optional.* A specific polling object for the engine to use.

`callback`(*function*, **args*, ***kwargs*)

Schedule a callback.

A callback is a function (or other callable) that is executed the next time *`poll()`* is called - in other words, on the next iteration of the main loop.

Returns a callable which can be used to cancel the callback.

Argument	Description
function	The callable to be executed when the callback is run.
args	The positional arguments to be passed to the callable.
kwargs	The keyword arguments to be passed to the callable.

`cycle`(*interval*, *function*, **args*, ***kwargs*)

Schedule a cycle.

A cycle is a deferred that is continuously rescheduled. It will be run at regular intervals.

Returns a callable which can be used to cancel the cycle.

Argument	Description
interval	The interval, in seconds, at which the cycle should be run.
function	The callable to be executed when the cycle is run.
args	The positional arguments to be passed to the callable.
kwargs	The keyword arguments to be passed to the callable.

`defer`(*delay*, *function*, **args*, ***kwargs*)

Schedule a deferred.

A deferred is a function (or other callable) that is executed after a certain amount of time has passed.

Returns a callable which can be used to cancel the deferred.

Argument	Description
delay	The delay, in seconds, after which the deferred should be run.
function	The callable to be executed when the deferred is run.
args	The positional arguments to be passed to the callable.
kwargs	The keyword arguments to be passed to the callable.

classmethod `instance`()
Returns the global engine object.

`loop`(*function*, **args*, ***kwargs*)
Schedule a loop.

A loop is a callback that is continuously rescheduled. It will be executed every time *`poll()`* is called - in other words, on each iteraton of the main loop.

Returns a callable which can be used to cancel the loop.

Argument	Description
function	The callable to be executed when the loop is run.
args	The positional arguments to be passed to the callable.
kwargs	The keyword arguments to be passed to the callable.

`poll`(*poll_timeout*)
Poll the engine.

Updates timers and processes I/O events on all active channels. If your application has a pre-existing main loop, call *`poll()`* on each iteration of that loop, otherwise, see *`start()`*.

Argument	Description
poll_timeout	The timeout to be passed to the polling object.

`start`(*poll_timeout=0.2*)
Start the engine.

Initialises and continuously polls the engine until either *`stop()`* is called or an uncaught `Exception` is raised. *`start()`* should be called after your asynchronous application has been fully initialised. For applications with a pre-existing main loop, see *`poll()`*.

Argument	Description
poll_timeout	*Optional.* The timeout to pass to *`poll()`*.

`stop`()
Stop the engine.

If *`start()`* has been called, calling *`stop()`* will cause the engine to cease polling and shut down on the next iteration of the main loop.

`pants.stream`

Streaming (TCP) connection implementation.

Streams are one of the two main types of channels in Pants - the other being *`servers`*. Streams represent connections between two endpoints. They may be used for both client and server applications.

Streams

To write a Pants application you will first need to subclass *`Stream`*. Your *`Stream`* subclass will contain the majority of your networking code in the form of event handlers. Event handlers are methods beginning with `on_` and can be safely overridden by your subclass.

Connecting

Before a *Stream* instance can be used, it must first be connected to a remote host. If you are writing a server application, all new *Stream* instance created by your *Server* will be connected. Once they are created by the *Server*, *on_connect()* will be called and your *Engine* will begin dispatching events to your *Stream* instance.

If you are writing a client application, you must first instantiate your *Stream* subclass and then use the *connect()* method to connect the channel to a remote host. Once the connection has been successfully established, the *on_connect()* event handler will be called and your *Stream* instance will start receiving events. Bear in mind that the connection will not be established until the *Engine* is running. As such, a common pattern when writing client applications with Pants is to call *connect()*, start the engine and then put all other initialization code in *on_connect()*.

Writing Data

Once your *Stream* instance is connected to a remote host, you can begin to write data to the channel. Use *write()* to write string data to the channel, *write_file()* to efficiently write data from an open file and *write_packed()* to write packed binary data. As you call these methods, Pants internally buffers your outgoing data. Once the buffer is completely empty, *on_write()* will be called. Be aware that if you continuously write data to your *Stream* that *on_write()* may not be called very frequently. If you wish to bypass the internal buffering and attempt to write your data immediately you can use the `flush` options present in the three write methods or call the *flush()* method yourself. This can help to improve your application's responsiveness but calling it excessively can reduce overall performance. Generally speaking, it is useful when you know with certainty that you have finished writing one discrete chunk of data (i.e. an HTTP response).

Reading Data

A connected *Stream* instance will automatically receive all incoming data from the remote host. By default, all incoming data is immediately passed to the *on_read()* event handler for your code to process. The *read_delimiter* attribute can be used to control this behaviour by causing Pants to buffer incoming data internally, only forwarding it to *on_read()* when a particular condition is met. If the condition is never met, the internal buffer will eventually exceed the allowed *buffer_size* and the *on_overflow_error()* handler method will be called. *read_delimiter* is extremely powerful when used effectively.

Closing

To close a *Stream* instance, simply call the *close()* method. Once a stream has been closed it should not be reused.

Handling Errors

Despite best efforts, errors will occasionally occur in asynchronous code. Pants handles these errors by passing the resulting exception object to one of a number of error handler methods. They are: *on_connect_error()*, *on_overflow_error()* and *on_error()*. Additionally, *on_ssl_handshake_error()* and *on_ssl_error()* exist to handle SSL-specific errors.

SSL

Pants streams have SSL support. If you are writing a server application, use *Server.startSSL* to enable SSL on your server. Each *Stream* created by your server from that point forward will be SSL-enabled. If you are writing

a client application, call *Stream.startSSL* before calling *connect()*. Alternatively, you can pass a dictionary of SSL options to the *Stream* constructor which will then enable SSL on the instance. When SSL is enabled on a *Stream*, an SSL handshake occurs between the local and remote ends of the connection. Once the SSL handshake is complete, *on_ssl_handshake()* will be called. If it fails, *on_ssl_handshake_error()* will be called.

If you are writing an SSL-enabled application you should read the entirety of Python's `ssl` documentation. Pants does not override any of Python's SSL defaults unless clearly stated in this documentation.

Stream

class `pants.stream.`**Stream**(***kwargs*)

The stream-oriented connection channel.

A *Stream* instance represents either a local connection to a remote server or a remote connection to a local server over a streaming, connection-oriented protocol such as TCP.

Keyword Argument	Description
engine	*Optional.* The engine to which the channel should be added. Defaults to the global engine.
socket	*Optional.* A pre-existing socket to wrap. This can be a regular `socket` or an `SSLSocket`. If a socket is not provided, a new socket will be created for the channel when required.
ssl_options	*Optional.* If provided, *startSSL()* will be called with these options once the stream is ready. By default, SSL will not be enabled.

buffer_size

The maximum size, in bytes, of the internal buffer used for incoming data.

When buffering data it is important to ensure that inordinate amounts of memory are not used. Setting the buffer size to a sensible value can prevent coding errors or malicious use from causing your application to consume increasingly large amounts of memory. By default, a maximum of 64kb of data will be stored.

The buffer size is mainly relevant when using a string value for the *read_delimiter*. Because you cannot guarantee that the string will appear, having an upper limit on the size of the data is appropriate.

If the read delimiter is set to a number larger than the buffer size, the buffer size will be increased to accommodate the read delimiter.

When the internal buffer's size exceeds the maximum allowed, the *on_overflow_error()* callback will be invoked.

Attempting to set the buffer size to anything other than an integer or long will raise a `TypeError`.

close(*flush=True*)

Close the channel.

connect(*address*)

Connect the channel to a remote socket.

The given `address` is resolved and used by the channel to connect to the remote server. If an error occurs at any stage in this process, *on_connect_error()* is called. When a connection is successfully established, *on_connect()* is called.

Addresses can be represented in a number of different ways. A single string is treated as a UNIX address. A single integer is treated as a port and converted to a 2-tuple of the form `('', port)`. A 2-tuple is treated as an IPv4 address and a 4-tuple is treated as an IPv6 address. See the `socket` documentation for further information on socket addresses.

If no socket exists on the channel, one will be created with a socket family appropriate for the given address.

An error will occur during the connection if the given address is not of a valid format or of an inappropriate format for the socket (e.g. if an IP address is given to a UNIX socket).

Calling *connect()* on a closed channel or a channel that is already connected will raise a `RuntimeError`.

Returns the channel.

Arguments	Description
address	The remote address to connect to.

flush()
Attempt to immediately write any internally buffered data to the channel without waiting for a write event.

This method can be fairly expensive to call and should be used sparingly.

Calling *flush()* on a closed or disconnected channel will raise a `RuntimeError`.

local_address
The address of the channel on the local machine.

By default, this will be the value of `socket.getsockname` or None. It is possible for user code to override the default behaviour and set the value of the property manually. In order to return the property to its default behaviour, user code then has to delete the value. Example:

```
# default behaviour
channel.local_address = custom_value
# channel.local_address will return custom_value now
del channel.local_address
# default behaviour
```

on_close()
Placeholder. Called after the channel has finished closing.

on_connect()
Placeholder. Called after the channel has connected to a remote socket.

on_connect_error(*exception*)
Placeholder. Called when the channel has failed to connect to a remote socket.

By default, logs the exception and closes the channel.

Argument	Description
exception	The exception that was raised.

on_error(*exception*)
Placeholder. Generic error handler for exceptions raised on the channel. Called when an error occurs and no specific error-handling callback exists.

By default, logs the exception and closes the channel.

Argument	Description
exception	The exception that was raised.

on_overflow_error(*exception*)
Placeholder. Called when an internal buffer on the channel has exceeded its size limit.

By default, logs the exception and closes the channel.

Argument	Description
exception	The exception that was raised.

on_read(*data*)
Placeholder. Called when data is read from the channel.

Argument	Description
data	A chunk of data received from the socket.

on_ssl_error(*exception*)
Placeholder. Called when an error occurs in the underlying SSL implementation.

By default, logs the exception and closes the channel.

Argument	Description
exception	The exception that was raised.

on_ssl_handshake()
Placeholder. Called after the channel has finished its SSL handshake.

on_ssl_handshake_error(*exception*)
Placeholder. Called when an error occurs during the SSL handshake.

By default, logs the exception and closes the channel.

Argument	Description
exception	The exception that was raised.

on_write()
Placeholder. Called after the channel has finished writing data.

read_delimiter
The magical read delimiter which determines how incoming data is buffered by the stream.

As data is read from the socket, it is buffered internally by the stream before being passed to the *on_read()* callback. The value of the read delimiter determines when the data is passed to the callback. Valid values are `None`, a byte string, an integer/long, a compiled regular expression, an instance of `struct.Struct`, or an instance of `netstruct.NetStruct`.

When the read delimiter is `None`, data will be passed to *on_read()* immediately after it is read from the socket. This is the default behaviour.

When the read delimiter is a byte string, data will be buffered internally until that string is encountered in the incoming data. All data up to but excluding the read delimiter is then passed to *on_read()*. The segment matching the read delimiter itself is discarded from the buffer.

When the read delimiter is an integer or a long, it is treated as the number of bytes to read before passing the data to *on_read()*.

When the read delimiter is a `struct.Struct` instance, the Struct's `size` is fully buffered and the data is unpacked using the Struct before its sent to *on_read()*. Unlike other types of read delimiters, this can result in more than one argument being passed to *on_read()*, as in the following example:

```
import struct
from pants import Stream

class Example(Stream):
    def on_connect(self):
        self.read_delimiter = struct.Struct("!LLH")

    def on_read(self, packet_type, length, id):
        pass
```

When the read delimiter is an instance of `netstruct.NetStruct`, the NetStruct's `minimum_size` is buffered and unpacked with the NetStruct, and additional data is buffered as necessary until the NetStruct can be completely unpacked. Once ready, the data will be passed to *on_read()*. Using Struct and NetStruct are *very* similar.

When the read delimiter is a compiled regular expression (`re.RegexObject`), there are two possible behaviors that you may switch between by setting the value of `regex_search`. If `regex_search` is True, as is the default, the delimiter's `search()` method is used and, if a match is found, the string before that match is passed to *`on_read()`*. The segment that was matched by the regular expression will be discarded.

If `regex_search` is False, the delimiter's `match()` method is used instead and, if a match is found, the match object itself will be passed to *`on_read()`*, giving you access to the capture groups. Again, the segment that was matched by the regular expression will be discarded from the buffer.

Attempting to set the read delimiter to any other value will raise a `TypeError`.

The effective use of the read delimiter can greatly simplify the implementation of certain protocols.

`remote_address`
The remote address to which the channel is connected.

By default, this will be the value of `socket.getpeername` or None. It is possible for user code to override the default behaviour and set the value of the property manually. In order to return the property to its default behaviour, user code then has to delete the value. Example:

```
# default behaviour
channel.remote_address = custom_value
# channel.remote_address will return custom_value now
del channel.remote_address
# default behaviour
```

`startSSL`(*ssl_options={}*)
Enable SSL on the channel and perform a handshake at the next opportunity.

SSL is only enabled on a channel once all currently pending data has been written. If a problem occurs at this stage, *`on_ssl_error()`* is called. Once SSL has been enabled, the SSL handshake begins - this typically takes some time and may fail, in which case *`on_ssl_handshake_error()`* will be called. When the handshake is successfully completed, *`on_ssl_handshake()`* is called and the channel is secure.

Typically, this method is called before *`connect()`*. In this case, *`on_ssl_handshake()`* will be called before *`on_connect()`*. If *`startSSL()`* is called after *`connect()`*, the reverse is true.

It is possible, although unusual, to start SSL on a channel that is already connected and active. In this case, as noted above, SSL will only be enabled and the handshake performed after all currently pending data has been written.

The SSL options argument will be passed through to `ssl.wrap_socket()` as keyword arguments - see the `ssl` documentation for further information. You will typically want to provide the `keyfile`, `certfile` and `ca_certs` options. The `do_handshake_on_connect` option **must** be `False`, or a `ValueError` will be raised.

Attempting to enable SSL on a closed channel or a channel that already has SSL enabled on it will raise a `RuntimeError`.

Returns the channel.

Arguments	Description
ssl_options	*Optional.* Keyword arguments to pass to `ssl.wrap_socket()`.

`write`(*data*, *flush=False*)
Write data to the channel.

Data will not be written immediately, but will be buffered internally until it can be sent without blocking the process.

Calling *write()* on a closed or disconnected channel will raise a RuntimeError.

Arguments	Description
data	A string of data to write to the channel.
flush	*Optional.* If True, flush the internal write buffer. See *flush()* for details.

write_file(*sfile*, *nbytes=0*, *offset=0*, *flush=False*)
Write a file to the channel.

The file will not be written immediately, but will be buffered internally until it can be sent without blocking the process.

Calling *write_file()* on a closed or disconnected channel will raise a RuntimeError.

Arguments	Description
sfile	A file object to write to the channel.
nbytes	*Optional.* The number of bytes of the file to write. If 0, all bytes will be written.
offset	*Optional.* The number of bytes to offset writing by.
flush	*Optional.* If True, flush the internal write buffer. See *flush()* for details.

write_packed(**data*, ***kwargs*)
Write packed binary data to the channel.

If the current *read_delimiter* is an instance of struct.Struct or netstruct.NetStruct the format will be read from that Struct, otherwise you will need to provide a format.

Argument	Description
*data	Any number of values to be passed through struct and written to the remote host.
flush	*Optional.* If True, flush the internal write buffer. See *flush()* for details.
format	*Optional.* A formatting string to pack the provided data with. If one isn't provided, the read delimiter will be used.

pants.server

Streaming (TCP) server implementation.

Servers are one of the two main types of channels in Pants - the other being *streams*. Servers listen for connections to your application, accept those connections and allow you to handle them easily. Pants servers support SSL and IPv6.

Servers

Writing Servers

You have two choices when writing a server application: either use Pants' default *Server* class without modification or subclass *Server* in order to implement custom behaviour.

Pants' default *Server* class will wrap every new connection in an instance of a connection class which you provide (see below). In most cases, this provides you with sufficient freedom to implement your application logic and has the added benefit of simplicity. To use the default server, simply instantiate *Server* and pass your connection class to the constructor.

If you need to implement custom server behaviour, you can subclass *Server* and define your connection class as a class attribute:

```
class MyServer(pants.Server):
    ConnectionClass = MyConnectionClass
```

It is recommended that you use the default *Server* class where possible and try to implement your application logic in your connection class.

Connection Classes

A connection class is a subclass of *Stream* which a server will use to wrap each incoming connection. Every time a new connection is made to the server, a new instance of your connection class will be created to handle it. You can override the various event handler methods of *Stream* to implement your application's logic.

Running Servers

Having defined your connection class and instantiated your server, you can start it listening for new connections with the *listen()* method. This will bind the server to your chosen address and once the *engine* is started, the server will begin accepting new connections. Once the server has started listening for connections it can be stopped using the *close()* method. When *close()* is called, the default server implementation will close any connections that were made to it which are still open.

SSL

Pants servers have SSL support. If you want to start an SSL-enabled server, call the *startSSL()* method before calling the *listen()* method. When you call *startSSL()* you must provide a dictionary of SSL options as detailed in the method documentation. It is also possible to pass the SSL options dictionary directly to the *Server* constructor in order to enable SSL. Here is an example of how you might start an SSL-enabled server:

```
server = pants.Server(MyConnectionClass)
server.startSSL({
    'certfile': '/home/user/certfile.pem',
    'keyfile': '/home/user/keyfile.pem'
    })
server.listen(('', 8080))
```

If you are writing an SSL-enabled application you should read the entirety of Python's `ssl` documentation. Pants does not override any of Python's SSL defaults unless clearly stated in this documentation.

Server

class `pants.server.`**`Server`**(*ConnectionClass=None*, ***kwargs*)

A stream-oriented server channel.

A *Server* instance represents a local server capable of listening for connections from remote hosts over a connection-oriented protocol such as TCP/IP.

Keyword Argument	Description
engine	*Optional.* The engine to which the channel should be added. Defaults to the global engine.
socket	*Optional.* A pre-existing socket to wrap. This can be a regular `socket` or an `SSLSocket`. If a socket is not provided, a new socket will be created for the channel when required.
ssl_options	*Optional.* If provided, *startSSL()* will be called with these options once the server is ready. By default, SSL will not be enabled.

close()
Close the channel.

The channel will be closed immediately and will cease to accept new connections. Any connections accepted by this channel will remain open and will need to be closed separately. If this channel has an IPv4 slave (see *listen()*) it will be closed.

Once closed, a channel cannot be re-opened.

listen(*address*, *backlog=1024*, *slave=True*)
Begin listening for connections made to the channel.

The given address is resolved, the channel is bound to the address and then begins listening for connections. Once the channel has begun listening, *on_listen()* will be called.

Addresses can be represented in a number of different ways. A single string is treated as a UNIX address. A single integer is treated as a port and converted to a 2-tuple of the form ('', port). A 2-tuple is treated as an IPv4 address and a 4-tuple is treated as an IPv6 address. See the socket documentation for further information on socket addresses.

If no socket exists on the channel, one will be created with a socket family appropriate for the given address.

An error will occur if the given address is not of a valid format or of an inappropriate format for the socket (e.g. if an IP address is given to a UNIX socket).

Calling *listen()* on a closed channel or a channel that is already listening will raise a RuntimeError.

Returns the channel.

Arguments	Description
address	The local address to listen for connections on.
backlog	*Optional.* The maximum size of the connection queue.
slave	*Optional.* If True, this will cause a Server listening on IPv6 INADDR_ANY to create a slave Server that listens on the IPv4 INADDR_ANY.

on_accept(*socket*, *addr*)
Called after the channel has accepted a new connection.

Create a new instance of ConnectionClass to wrap the socket and add it to the server.

Argument	Description
sock	The newly connected socket object.
addr	The new socket's address.

on_close()
Called after the channel has finished closing.

Close all active connections to the server.

on_error(*exception*)
Placeholder. Generic error handler for exceptions raised on the channel. Called when an error occurs and no specific error-handling callback exists.

By default, logs the exception and closes the channel.

Argument	Description
exception	The exception that was raised.

on_listen()
Placeholder. Called when the channel begins listening for new connections or packets.

on_ssl_wrap_error(*sock*, *addr*, *exception*)
Placeholder. Called when an error occurs while wrapping a new connection with an SSL context.

By default, logs the exception and closes the new connection.

Argument	Description
sock	The newly connected socket object.
addr	The new socket's address.
exception	The exception that was raised.

startSSL(*ssl_options={}*)
Enable SSL on the channel.

Enabling SSL on a server channel will cause any new connections accepted by the server to be automatically wrapped in an SSL context before being passed to *on_accept()*. If an error occurs while a new connection is being wrapped, *on_ssl_wrap_error()* is called.

SSL is enabled immediately. Typically, this method is called before *listen()*. If it is called afterwards, any connections made in the meantime will not have been wrapped in SSL contexts.

The SSL options argument will be passed through to each invocation of `ssl.wrap_socket()` as keyword arguments - see the `ssl` documentation for further information. You will typically want to provide the `keyfile`, `certfile` and `ca_certs` options. The `do_handshake_on_connect` option **must** be `False` and the `server_side` option **must** be true, or a `ValueError` will be raised.

Attempting to enable SSL on a closed channel or a channel that already has SSL enabled on it will raise a `RuntimeError`.

Returns the channel.

Arguments	Description
ssl_options	*Optional.* Keyword arguments to pass to `ssl.wrap_socket()`.

HTTP & Web

pants.http.server

`pants.http.server` implements a lean HTTP server on top of Pants with support for most of HTTP/1.1, including persistent connections. The HTTP server supports secure connections, efficient transfer of files, and proxy headers. Utilizing the power of Pants, it becomes easy to implement other protocols on top of HTTP such as *WebSockets*.

The Server

HTTPServer is a subclass of *pants.server.Server* that implements the HTTP/1.1 protocol via the class *HTTPConnection*. Rather than specifying a custom `ConnectionClass`, you implement your behavior with a `request_handler`. There will be more on request handlers below. For now, a brief example:

```
from pants.http import HTTPServer
from pants import Engine

def my_handler(request):
    request.send_response("Hello World.")

server = HTTPServer(my_handler)
server.listen()
Engine.instance().start()
```

In addition to specifying the request handler, there are a few other ways to configure `HTTPServer`.

Using HTTPServer Behind a Proxy

`HTTPServer` has support for a few special HTTP headers that can be set by proxy servers (notably `X-Forwarded-For` and `X-Forwarded-Proto`) and it can use `X-Sendfile` headers when sending files to allow the proxy server to take care of static file transmission.

When creating your *`HTTPServer`* instance, set `xheaders` to `True` to allow the server to automatically use the headers `X-Real-IP`, `X-Forwarded-For`, and `X-Forwarded-Proto` if they exist to set the *`HTTPRequest`*'s `remote_ip` and `scheme`.

Sendfile is a bit more complex, with three separate variables for configuration. To enable the `X-Sendfile` header, set `sendfile` to `True` when creating your *`HTTPServer`* instance. Alternatively, you may set it to a string to have Pants use a string other than `X-Sendfile` for the header's name.

HTTPServer's `sendfile_prefix` allows you to set a prefix for the path written to the `X-Sendfile` header. This is useful when using Pants behind nginx.

HTTPServer's `file_root` allows you to specify a root directory from which static files should be located. This root path will be stripped from the file paths before they're written to the `X-Sendfile` header. If `file_root` is not set, the current working directory (as of the time *`HTTPRequest.send_file()`* is called) will be used.

```
def my_handler(request):
    request.send_file('/srv/files/example.jpg')

server = HTTPServer(my_handler, sendfile=True, sendfile_prefix='/_static/',
                    file_root='/srv/files')
server.listen()
```

The above code would result in an HTTP response similar to:

```
HTTP/1.1 200 OK
Content-Type: image/jpeg
Content-Length: 0
X-Sendfile: /_static/example.jpg
```

Your proxy server would then be required to detect the `X-Sendfile` header in that response and insert the appropriate content and headers.

Note: The sendfile API is quite rough at this point, and is most likely going to be changed in future versions. It is possible to manually set the appropriate headers to handle sending files yourself if you require more control over the process.

Request Handlers

A request handler is a callable Python object, typically either a function or a class instance with a defined `__call__` method. Request handlers are passed an instance of *`HTTPRequest`* representing the current request.

HTTPRequest instances contain all of the information that was sent with an incoming request. The instances also have numerous methods for building responses.

Note: It is *not* required to finish responding to a request within the request handler.

Please see the documentation for the *HTTPRequest* class below for more information on what you can do.

HTTPServer

class pants.http.server.**HTTPServer**(*request_handler*, *max_request=10485760*, *keep_alive=True*, *cookie_secret=None*, *xheaders=False*, *sendfile=False*, *sendfile_prefix=None*, *file_root=None*, ***kwargs*)

An HTTP server, extending the default *Server* class.

This class automatically uses the *HTTPConnection* connection class. Rather than through specifying a connection class, its behavior is customized by providing a request handler function that is called whenever a valid request is received.

A server's behavior is defined almost entirely by its request handler, and will not send any response by itself unless the received HTTP request is not valid or larger than the specified limit (which defaults to 10 MiB, or 10,485,760 bytes).

Argument	Default	Description
request_handler		A callable that accepts a single argument. That argument is an instance of the *HTTPRequest* class representing the current request.
max_request	10 MiB	*Optional.* The maximum allowed length, in bytes, of an HTTP request body. This should be kept small, as the entire request body will be held in memory.
keep_alive	True	*Optional.* Whether or not multiple requests are allowed over a single connection.
cookie_secret	None	*Optional.* A string to use when signing secure cookies.
xheaders	False	*Optional.* Whether or not to use `X-Forwarded-For` and `X-Forwarded-Proto` headers.
sendfile	False	*Optional.* Whether or not to use `X-Sendfile` headers. If this is set to a string, that string will be used as the header name.
sendfile_prefix	None	*Optional.* A string to prefix paths with for use in the `X-Sendfile` headers. Useful for nginx.
file_root	None	*Optional.* The root path to send files from using *send_file()*.

startSSL(*ssl_options={}*)

Enable SSL on the server, creating an `HTTPS` server.

When an HTTP server has been secured, the `scheme` of all *HTTPRequest* instances is set to `https`, otherwise it will be `http`. Please note that the `X-Forwarded-Proto` may override `scheme` if `xheaders` is set to `True`.

See also:

See *pants.server.Server.startSSL()* for more information on how SSL is implemented within Pants.

listen(*address=None*, *backlog=1024*, *slave=True*)

Begins listening for connections to the HTTP server.

The given `address` is resolved, the server is bound to the address, and it then begins listening for connections. If an address isn't specified, the server will listen on either port 80 or port 443 by default. Port 443 is selected if SSL has been enabled prior to the call to listen, otherwise port 80 will be used.

See also:

See *pants.server.Server.listen()* for more information on listening servers.

HTTPConnection

class pants.http.server.**HTTPConnection**(**args*, ***kwargs*)

This class implements the HTTP protocol on top of Pants. It specifically processes incoming HTTP requests and constructs an instance of *HTTPRequest* before passing that instance to the associated *HTTPServer*'s request handler.

Direct interaction with this class is typically unnecessary, only becoming useful when implementing another protocol on top of HTTP, such as *WebSockets* or performing some other action that requires direct control over the underlying socket.

current_request

An instance of *HTTPRequest* representing the active request on the connection. If there is no active request, this will be None.

finish()

This method should be called when the response to the current request has been completed, in preparation for either closing the connection or attempting to read a new request from the connection.

This method is called automatically when you use the method *HTTPRequest.finish()*.

HTTPRequest

class pants.http.server.**HTTPRequest**(*connection*, *method*, *url*, *protocol*, *headers=None*, *scheme='http'*)

Instances of this class represent single HTTP requests that an *HTTPServer* has received. Such instances contain all the information needed to respond to the request, as well as the functions used to build the appropriate response.

HTTPRequest uses bytes rather than str unless otherwise stated, as network communications take place as bytes.

remote_ip

The IP address of the client, represented as bytes. If the underlying *HTTPServer* is set to use xheaders, this value may be loaded from the X-Real-Ip or X-Forwarded-For headers.

scheme

The scheme through which the request was received. This will typically be http. The scheme will be set to https when the connection the request was received across is secured. If the underlying *HTTPServer* is set to use xheaders, this value may be loaded from the X-Forwarded-Proto header.

protocol

The protocol the request was received across. This is typically either HTTP/1.1 or HTTP/1.0.

method

The HTTP request method, such as GET or POST.

url

The URL that has been requested.

path

The path segment of the *url*. Note that Pants does not separate the path and parameters segments automatically to save time on each request as the parameters segment is not often utilized.

query

The query segment of the *url*.

fragment

The fragment segment of the *url*.

headers
An instance of `pants.http.utils.HTTPHeaders` containing the headers that were received with the request. HTTPHeaders is effectively a case-insensitive dictionary that normalizes header cases upon iteration.

host
The host that the request was directed to. This is, effectively, the value of the request's `Host` header. If no such header exists, the value will be set to the bytes `127.0.0.1`.

Unlike the *`hostname`*, this value may contain a port number if the request was sent to a non-standard port.

hostname
The hostname segment of the *`host`*. This value will always be lower-case.

get
A dictionary of HTTP GET variables. The variables are parsed from the *`query`* using `urlparse.parse_qsl()` with `keep_blank_values` set to `False`.

post
A dictionary of HTTP POST variables. For security, this variable is *only* populated if the *`method`* is `POST` or `PUT`.

If the request's `Content-Type` header is set to `application/x-www-form-urlencoded`, the variables will be parsed from the *`body`* using `urlparse.parse_sql()` with `keep_blank_values` set to `False`.

If the request's `Content-Type` header is set to `multipart/form-data`, the *`body`* will be processed for both POST variables and *`files`*.

files
A dictionary containing files received within the request body. For security, this variable is *only* populated if the *`method`* is `POST` or `PUT`. At this time, Pants only knows how to receive files when the request body is formatted as `multipart/form-data`.

The form data variable names will be used for the dictionary keys. Each key will contain a list with one or more dictionaries representing the received files. A file's dictionary has the keys: `filename`, `body`, and `content_type`.

You might receive a file using:

```
def my_handler(request):
    contents = request.files['my_field'][0]['body']
```

Note: Pants does a poor job of handling files at this time, keeping them entirely in memory while a request is being handled. It is recommended to use a proxy server with some way to receive files when writing applications.

In the future, the Pants HTTP server will be modified so that large request bodies and received files are stored to disk as temporary files as they're received to reduce memory utilization.

---

body
A `bytes` instance containing the entire request body that has not been processed in any way.

connection
The underlying *`HTTPConnection`* instance that received this request. You shouldn't have need to use this in most situations.

cookies
An instance of `Cookie.SimpleCookie` representing the cookies received with this request. Cookies being sent to the client with the response are stored in *`cookies_out`*.

cookies_out
An instance of `Cookie.SimpleCookie` to populate with cookies that should be sent with the response.

finish()
This function should be called when the response has been completed, allowing the associated *`HTTPConnection`* to either close the connection to the client or begin listening for a new request.

Failing to call this function will drastically reduce the performance of the HTTP server, if it will work at all.

full_url
The full url for this request. This is created by combining the *`scheme`*, *`host`*, and the *`url`*.

get_secure_cookie(*name*)
Return the signed cookie with the key `name` if it exists and has a valid signature. Otherwise, return None.

is_secure
Whether or not the request was received via HTTPS.

send(*data*)
Write data to the client.

Argument	Description
data	A string of data to be sent to the client.

send_cookies(*keys=None*, *end_headers=False*)
Write any cookies associated with the request to the client. If any keys are specified, only the cookies with the specified keys will be transmitted. Otherwise, all cookies in *`cookies_out`* will be written to the client.

This function is usually called automatically by send_headers.

Argument	Default	Description
keys	None	*Optional.* A list of cookie names to send.
end_headers	False	*Optional.* If this is set to True, a double CRLF sequence will be written at the end of the cookie headers, signifying the end of the HTTP headers segment and the beginning of the response.

send_file(*path*, *filename=None*, *guess_mime=True*, *headers=None*)
Send a file to the client, given the path to that file. This method makes use of `X-Sendfile`, if the *`HTTPServer`* instance is configured to send X-Sendfile headers.

If `X-Sendfile` is not available, Pants will make full use of caching headers, Ranges, and the sendfile system call to improve file transfer performance. Additionally, if the client had made a `HEAD` request, the contents of the file will not be transferred.

Note: The request is finished automatically by this method.

Argument	Default	Description
path		The path to the file to send. If this is a relative path, and the *HTTPServer* instance has no root path for Sendfile set, the path will be assumed relative to the current working directory.
filename	None	*Optional.* If this is set, the file will be sent as a download with the given filename as the default name to save it with.
guess_mime	True	*Optional.* If this is set to True, Pants will attempt to set the `Content-Type` header based on the file extension.
headers	None	*Optional.* A dictionary of HTTP headers to send with the file.

Note: If you set a `Content-Type` header with the `headers` parameter, the mime type will not be used, even if `guess_mime` is True. The `headers` will also override any `Content-Disposition` header generated by the `filename` parameter.

send_headers(*headers*, *end_headers=True*, *cookies=True*)

Write a dictionary of HTTP headers to the client.

Argument	Default	Description
headers		A dictionary of HTTP headers.
end_headers	True	*Optional.* If this is set to True, a double CRLF sequence will be written at the end of the cookie headers, signifying the end of the HTTP headers segment and the beginning of the response.
cookies	True	*Optional.* If this is set to True, HTTP cookies will be sent along with the headers.

send_response(*content*, *code=200*, *content_type='text/plain'*)

Write a very simple response, in one easy function. This function is for convenience, and allows you to send a basic response in one line.

Basically, rather than:

```
def request_handler(request):
    output = "Hello, World!"

    request.send_status(200)
    request.send_headers({
        'Content-Type': 'text/plain',
        'Content-Length': len(output)
        })
    request.send(output)
    request.finish()
```

You can simply:

```
def request_handler(request):
    request.send_response("Hello, World!")
```

Argument	Default	Description
content		A string of content to send to the client.
code	`200`	*Optional.* The HTTP status code to send to the client.
content_type	`text/plain`	*Optional.* The Content-Type header to send.

send_status(*code=200*)
Write an HTTP status line (the very first line of any response) to the client, using the same HTTP protocol version as the request. If one is available, a human readable status message will be appended after the provided code.

For example, `request.send_status(404)` would result in `HTTP/1.1 404 Not Found` being sent to the client, assuming of course that the request used HTTP protocol version `HTTP/1.1`.

Argument	Default	Description
code	`200`	*Optional.* The HTTP status code to send to the client.

set_secure_cookie(*name*, *value*, *expires=2592000*, ***kwargs*)
Set a timestamp on a cookie and sign it, ensuring that it can't be altered by the client. To use this, the *`HTTPServer`* *must* have a `cookie_secret` set.

Cookies set with this function may be read with *`get_secure_cookie()`*.

If the provided value is a dictionary, list, or tuple the value will be serialized into JSON and encoded as UTF-8. Unicode strings will also be encoded as UTF-8. Byte strings will be passed as is. All other types will result in a `TypeError`.

Argument	Default	Description
name		The name of the cookie to set.
value		The value of the cookie.
expires	`2592000`	*Optional.* How long, in seconds, the cookie should last before expiring. The default value is equivalent to 30 days.

Additional arguments, such as `path` and `secure` may be set by providing them as keyword arguments. The `HttpOnly` attribute will be set by default on secure cookies..

time
The amount of time that has elapsed since the request was received. If the request has been finished already, this will be the total time that elapsed over the duration of the request.

pants.http.client

`pants.http.client` implements a basic asynchronous HTTP client on top of Pants with an API modelled after that of the wonderful requests library. The client supports keep-alive and SSL for connections, domain verification for SSL certificates, basic WWW authentication, sessions with persistent cookies, automatic redirect handling, automatic decompression of responses, connection timeouts, file uploads, and saving large responses to temporary files to decrease memory usage.

Logic is implemented using a series of request handlers.

Making Requests

It's simple and easy to make requests, and it only requires that you have an instance of *`HTTPClient`* ready.

```
from pants.http import HTTPClient
client = HTTPClient()
```

Like with requests, there are simple methods for making requests with the different HTTP methods. For now, let's get information for a bunch of Pants' commits on GitHub.

```
client.get("https://api.github.com/repos/ecdavis/pants/commits")
```

You'll notice that this is very similar to making a request with requests. However, we do not get a response objects. Actually, calling *HTTPClient.get()* returns an instance of *HTTPRequest* rather than anything to do with a response, but we'll get to that later.

The Pants HTTP client is asynchronous, so to get your response, you need a response handler. There are several ways to set one up, but the easiest way is to pass it to your *HTTPClient* during initialization.

```
def handle_response(response):
    if response.status_code != 200:
        print "There was a problem!"

client = HTTPClient(handle_response)
```

`response` in this situation is an instance of *HTTPResponse*, and it has an API modelled after the response objects that requests would give you.

Making *Useful* Requests

Basic GET requests are nice, but you'll often want to send data to the server. For query parameters you can use the optional *params* argument of the various request methods, like so:

```
data = {'since': '2013-11-01'}
client.get("https://api.github.com/repos/ecdavis/pants/commits", params=data)
```

With that, you could eventually take your response and get the correct URL.

```
>>> response.url
'https://api.github.com/repos/ecdavis/pants/commits?since=2013-11-01'
```

You can also post data to the server, either as a pre-made string, or as a dictionary of values to be encoded.

```
client.post("http://httpbin.org/post", data="Hello World!")
client.post("http://httpbin.org/post", data={"greeting": "Hello"})
```

By default, the `Content-Type` header will be set to `application/x-www-form-urlencoded` when you provide data for the request body. If any files are present, it will instead default to `multipart/form-data` to transmit those. You can also manually set the header when making your request.

You set files via the files parameter, which expects a dictionary of form field names and file objects. You can also provide filenames if desired.

```
client.post("http://httpbin.org/post", files={'file': open("test.txt")})
client.post("http://httpbin.org/post", files={'file': ("test.txt", open("test.txt"))})
```

You can, of course, use data and files together. Please note that, if you *do* use them together, you'll need to supply data as a dictionary. Data strings are not supported.

As many of you have probably noticed, this is *very* similar to using requests. The Pants API was implemented this way to make it easier to switch between the two libraries.

Reading Responses

Making your request is only half the battle, of course. You have to read your response when it comes in. And, for that, you start with the status code.

```
>>> response.status_code
200
>>> response.status_text
'OK'
>>> response.status
'200 OK'
```

Unlike with requests, there is no `raise_for_status()` method available. Raising a strange exception in an asynchronous framework that your code isn't designed to catch just wouldn't work.

Headers

HTTP headers are case-insensitive, and so the headers are stored in a special case-insensitive dictionary made available as *`HTTPResponse.headers`*.

```
>>> response.headers
HTTPHeaders({
    'Content-Length': 986,
    'Server': 'gunicorn/0.17.4',
    'Connection': 'keep-alive',
    'Date': 'Wed, 06 Nov 2013 05:58:53 GMT',
    'Access-Control-Allow-Origin': '*',
    'Content-Type': 'application/json'
    })
>>> response.headers['content-length']
986
```

Nothing special here.

Cookies

Cookies are a weak point of Pants' HTTP client at this time. Cookies are stored in instances of `Cookie.SimpleCookie`, which doesn't handle multiple domains. Pants has logic to prevent sending cookies to the wrong domains, but ideally it should move to using a better cookie storage structure in future versions that handles multiple domains elegantly.

```
>>> response.cookies['cake']
<Morsel: cake='lie'>
>>> response.cookies['cake'].value
'lie'
```

As you can see, Pants does not yet handle cookies as well as requests. Setting cookies is a bit better.

```
client.get("http://httpbin.org/cookies", cookies={"cake": "lie"})
```

Redirects

The HTTP client will follow redirects automatically. When this happens, the redirecting responses are stored in the `HTTPResponse.history` list.

```
>>> response.history
[<HTTPResponse [301 Moved Permanently] at 0x2C988F0>]
```

You can limit the number of times the HTTP client will automatically follow redirects with the `max_redirects` argument.

```
client.get("http://github.com/", max_redirects=0)
```

By default, Pants will follow up to 10 redirects.

Exceptions

class `pants.http.client.`**`CertificateError`**

class `pants.http.client.`**`HTTPClientException`**
The base exception for all the exceptions used by the HTTP client, aside from *`CertificateError`*.

class `pants.http.client.`**`MalformedResponse`**
The exception returned when the response is malformed in some way.

class `pants.http.client.`**`RequestClosed`**
The exception returned when the connection closes before the entire request has been downloaded.

class `pants.http.client.`**`RequestTimedOut`**
The exception returned when a connection times out.

HTTPClient

class `pants.http.client.`**`HTTPClient`**(**args*, ***kwargs*)
An easy to use, asynchronous HTTP client implementing HTTP 1.1. All arguments passed to HTTPClient are used to initialize the default session. See *`Session`* for more details. The following is a basic example of using an HTTPClient to fetch a remote resource:

```
from pants.http import HTTPClient
from pants.engine import Engine

def response_handler(response):
    Engine.instance().stop()
    print response.content

client = HTTPClient(response_handler)
client.get("http://httpbin.org/ip")
Engine.instance().start()
```

Groups of requests can have their behavior customized with the use of sessions:

```
from pants.http import HTTPClient
from pants.engine import Engine

def response_handler(response):
    Engine.instance().stop()
    print response.content

def other_handler(response):
    print response.content

client = HTTPClient(response_handler)
client.get("http://httpbin.org/cookies")

with client.session(cookies={'pie':'yummy'}):
```

```
    client.get("http://httpbin.org/cookies")

Engine.instance().start()
```

delete(*url*, ***kwargs*)
Begin a DELETE request. See *request()* for more details.

get(*url*, *params=None*, ***kwargs*)
Begin a GET request. See *request()* for more details.

head(*url*, *params=None*, ***kwargs*)
Begin a HEAD request. See *request()* for more details.

on_error(*response*, *exception*)
Placeholder. Called when an error occurs.

Argument	Description
exception	An Exception instance with information about the error that occurred.

on_headers(*response*)
Placeholder. Called when we've received headers for a request. You can abort a request at this time by returning False from this function. It *must* be False, and not simply a false-like value, such as an empty string.

Note: This function isn't called for HTTP `HEAD` requests.

Argument	Description
response	A *HTTPResponse* instance with information about the received response.

on_progress(*response*, *received*, *total*)
Placeholder. Called when progress is made in downloading a response.

Argument	Description
response	A *HTTPResponse* instance with information about the response.
received	The number of bytes received thus far.
total	The total number of bytes expected for the response. This will be `0` if we don't know how much to expect.

on_response(*response*)
Placeholder. Called when a complete response has been received.

Argument	Description
response	A *HTTPResponse* instance with information about the received response.

on_ssl_error(*response*, *certificate*, *exception*)
Placeholder. Called when the remote server's SSL certificate failed initial verification. If this method returns True, the certificate will be accepted, otherwise, the connection will be closed and *on_error()* will be called.

Argument	Description
response	A *HTTPResponse* instance with information about the response. Notably, with the `host` to expect.
certificate	A dictionary representing the certificate that wasn't automatically verified.
exception	A CertificateError instance with information about the error that occurred.

options(*url*, ***kwargs*)
Begin an OPTIONS request. See *request()* for more details.

patch(*url*, *data=None*, ***kwargs*)
Begin a PATCH request. See *request()* for more details.

post(*url*, *data=None*, *files=None*, ***kwargs*)
Begin a POST request. See *request()* for more details.

put(*url*, *data=None*, ***kwargs*)
Begin a PUT request. See *request()* for more details.

request(**args*, ***kwargs*)
Begin a request. Missing parameters will be taken from the active session when available. See *Session.request()* for more details.

session(**args*, ***kwargs*)
Create a new session. See *Session* for details.

trace(*url*, ***kwargs*)
Begin a TRACE request. See *request()* for more details.

HTTPRequest

class pants.http.client.**HTTPRequest**(*session*, *method*, *path*, *url*, *headers*, *cookies*, *body*, *timeout*, *max_redirects*, *keep_alive*, *auth*)
A very basic structure for storing HTTP request information.

response
The *HTTPResponse* instance representing the response to this request.

session
The *Session* this request was made in.

method
The HTTP method of this request, such as `GET`, `POST`, or `HEAD`.

path
The path of this request.

url
A tuple containing the full URL of the request, as processed by `urlparse.urlparse()`.

headers
A dictionary of headers sent with this request.

cookies
A `Cookie.SimpleCookie` instance of cookies sent with this request.

body
A list of strings and files sent as this request's body.

timeout
The time to wait, in seconds, of no activity to allow before timing out.

max_redirects
The maximum remaining number of redirects before not automatically redirecting.

keep_alive
Whether or not the connection should be reused after this request.

auth
Either a tuple of `(username, password)` or an instance of `AuthBase` responsible for authorizing this request with the server.

HTTPResponse

class `pants.http.client.`**HTTPResponse**(*request*)
The HTTPResponse class represents a single HTTPResponse, and has all the available information about a response, including the redirect history and the original HTTPRequest.

length
The length of the raw response.

http_version
The HTTP version of the response.

status_code
The HTTP status code of the response, such as `200`.

status_text
The human readable status text explaining the status code, such as `Not Found`.

cookies
A `Cookie.SimpleCookie` instance of all the cookies received with the response.

headers
A dictionary of all the headers received with the response.

content
The content of the response as a byte string. Be careful when using this with large responses, as it will load the entire response into memory. `None` if no data has been received.

encoding
This is the detected character encoding of the response. You can also set this to a specific character set to have *`text`* decoded properly.

Pants will attempt to fill this value from the Content-Type response header. If no value was available, it will be `None`.

file
The content of the response as a `tempfile.SpooledTemporaryFile`. Pants uses temporary files to decrease memory usage for large responses. `None` if no data has been received.

handle_301(*client*)
Handle the different redirect codes.

handle_401(*client*)
Handle authorization, if we know how.

iter_content(*chunk_size=1*, *decode_unicode=False*)
Iterate over the content of the response. Using this, rather than *`content`* or *`text`* can prevent the loading of large responses into memory in their entirety.

Argument	Default	Description
chunk_size	`1`	The number of bytes to read at once.
decode_unicode	False	Whether or not to decode the bytes into unicode using the response's *`encoding`*.

iter_lines(*chunk_size=512*, *decode_unicode=False*)
Iterate over the content of the response, one line at a time. By using this rather than *content* or *text* you can prevent loading of the entire response into memory. The two arguments to this method are passed directly to *iter_content()*.

json(***kwargs*)
The content of the response, having been interpreted as JSON. This uses the value of *encoding* if possible. If *encoding* is not set, it will default to `UTF-8`.

Any provided keyword arguments will be passed to `json.loads()`.

status
The status code and status text as a string.

text
The content of the response, after being decoded into unicode with *encoding*. Be careful when using this with large responses, as it will load the entire response into memory. `None` if no data has been received.

If *encoding* is `None`, this will default to `UTF-8`.

Session

class `pants.http.client.`**Session**(*client*, *on_response=None*, *on_headers=None*, *on_progress=None*, *on_ssl_error=None*, *on_error=None*, *timeout=None*, *max_redirects=None*, *keep_alive=None*, *auth=None*, *headers=None*, *cookies=None*, *verify_ssl=None*, *ssl_options=None*)
The Session class is the heart of the HTTP client, making it easy to share state between multiple requests, and enabling the use of `with` syntax. They're responsible for determining everything about a request before handing it back to *HTTPClient* to be executed.

Argument	Default	Description
client		The *HTTPClient* instance this Session is associated with.
on_response		*Optional.* A callable that will handle any received responses, rather than the HTTPClient's own `on_response()` method.
on_headers		*Optional.* A callable for when response headers have been received.
on_progress		*Optional.* A callable for progress notifications.
on_ssl_error		*Optional.* A callable responsible for handling SSL verification errors, if `verify_ssl` is True.
on_error		*Optional.* A callable that will handle any errors that occur.
timeout	`30`	*Optional.* The time to wait, in seconds, of no activity to allow before timing out.
max_redirects	`10`	*Optional.* The maximum number of times to follow a server-issued redirect.
keep_alive	`True`	*Optional.* Whether or not a single connection will be reused for multiple requests.
auth	`None`	*Optional.* An instance of `AuthBase` for authenticating requests to the server.
headers	`None`	*Optional.* A dictionary of default headers to send with requests.
verify_ssl	`False`	*Optional.* Whether or not to attempt to check the certificate of the remote secure server against its hostname.
ssl_options	`None`	*Optional.* Options to use when initializing SSL. See `Stream.startSSL()` for more.

client
The *HTTPClient* this Session is associated with.

delete(*url*, ***kwargs*)
Begin a DELETE request. See *request()* for more details.

get(*url*, *params=None*, ***kwargs*)
Begin a GET request. See `request()` for more details.

head(*url*, *params=None*, ***kwargs*)
Begin a HEAD request. See `request()` for more details.

options(*url*, ***kwargs*)
Begin an OPTIONS request. See `request()` for more details.

patch(*url*, *data=None*, ***kwargs*)
Begin a PATCH request. See `request()` for more details.

post(*url*, *data=None*, *files=None*, ***kwargs*)
Begin a POST request. See `request()` for more details.

put(*url*, *data=None*, ***kwargs*)
Begin a PUT request. See `request()` for more details.

request(*method*, *url*, *params=None*, *data=None*, *headers=None*, *cookies=None*, *files=None*, *auth=None*, *timeout=None*, *max_redirects=None*, *keep_alive=None*)
Begin a request.

Argument	Description
method	The HTTP method of the request.
url	The URL to request.
params	*Optional.* A dictionary or string of query parameters to add to the request.
data	*Optional.* A dictionary or string of content to send in the request body.
headers	*Optional.* A dictionary of headers to send with the request.
cookies	*Optional.* A dictionary or CookieJar of cookies to send with the request.
files	*Optional.* A dictionary of file-like objects to upload with the request.
auth	*Optional.* An instance of `AuthBase` to use to authenticate the request.
timeout	*Optional.* The time to wait, in seconds, of no activity to allow before timing out.
max_redirects	*Optional.* The maximum number of times to follow a server-issued redirect.
keep_alive	*Optional.* Whether or not to reuse the connection for multiple requests.

session(**args*, ***kwargs*)
Create a new session. See `Session` for details.

trace(*url*, ***kwargs*)
Begin a TRACE request. See `request()` for more details.

pants.http.websocket

`pants.http.websocket` implements the WebSocket protocol, as described by **RFC 6455**, on top of the Pants HTTP server using an API similar to that provided by `pants.stream.Stream`.

Using WebSockets

To start working with WebSockets, you'll need to create a subclass of `WebSocket`. As with `Stream`, `WebSocket` instances are meant to contain the majority of your networking logic through the definition of custom event handlers. Event handlers are methods that have names beginning with `on_` that can be safely overridden within your subclass.

Listening for Connections

`WebSocket` is designed to be used as a request handler for the Pants HTTP server, `pants.http.server.HTTPServer`. As such, to begin listening for WebSocket connections, you must create an instance of `HTTPServer`

using your custom *WebSocket* subclass as its request handler.

```
from pants.http import HTTPServer, WebSocket
from pants import Engine

class EchoSocket(WebSocket):
    def on_read(self, data):
        self.write(data)

HTTPServer(EchoSocket).listen(8080)
Engine.instance().start()
```

If you need to host traditional requests from your HTTPServer instance, you may invoke the WebSocket handler simply by creating an instance of your *WebSocket* subclass with the appropriate *pants.http.server.HTTPRequest* instance as its only argument:

```
from pants.http import HTTPServer, WebSocket
from pants import Engine

class EchoSocket(WebSocket):
    def on_read(self, data):
        self.write(data)

def request_handler(request):
    if request.path == '/_ws':
        EchoSocket(request)
    else:
        request.send_response("Nothing to see here.")

HTTPServer(request_handler).listen(8080)
Engine.instance().start()
```

WebSocket and Application

WebSocket has support for *pants.web.application.Application* and can easily be used as a request handler for any route. Additionally, variables captured from the URL by *Application* will be made accessible to the *WebSocket.on_connect()* event handler. The following example of a WebSocket echo server displays a customized welcome message depending on the requested URL.

```
from pants.http import WebSocket
from pants.web import Application

app = Application()

@app.route("/ws/<name>")
class EchoSocket(WebSocket):
    def on_connect(self, name):
        self.write(u"Hello, {name}!".format(name=name))

    def on_read(self, data):
        self.write(data)

app.run(8080)
```

WebSocket Security

Secure Connections

WebSocket relies upon the *pants.http.server.HTTPServer* instance serving it to provide SSL. This can be as easy as calling the server's *startSSL()* method.

To determine whether or not the *WebSocket* instance is using a secure connection, you may use the *is_secure* attribute.

Custom Handshakes

You may implement custom logic during the WebSocket's handshake by overriding the *WebSocket.on_handshake()* event handler. The `on_handshake` handler is called with a reference to the *HTTPRequest* instance the WebSocket handshake is happening upon as well as an empty dictionary that may be used to set custom headers on the HTTP response.

`on_handshake` is expected to return a True value if the request is alright. Returning a False value will result in the generation of an error page. The following example of a custom handshake requires a secret HTTP header in the request, and that the connection is secured:

```
from pants.http import WebSocket

class SecureSocket(WebSocket):
    def on_handshake(self, request, headers):
        return self.is_secure and 'X-Pizza' in request.headers

    def on_connect(self):
        self.write(u"Welcome to PizzaNet.")
```

Reading and Writing Data

WebSockets are a bit different than normal *Stream* instances, as a WebSocket can transmit both byte strings and unicode strings, and data is encapsulated into formatted messages with definite lengths. Because of this, reading from one can be slightly different.

Mostly, however, the *read_delimiter* works in exactly the same way as that of *pants.stream.Stream*.

Unicode Strings and Byte Strings

WebSocket strictly enforces the difference between byte strings and unicode strings. As such, the connection will be closed with a protocol error if any of the following happen:

1. The string types of the *read_delimiter* and the buffer differ.
2. There is an existing string still in the buffer when the client sends another string of a different type.
3. The *read_delimiter* is currently a struct and the buffer does not contain a byte string.

Of those conditions, the most likely to occur is the first. Take the following code:

```
from pants.http import WebSocket, HTTPServer
from pants import Engine

def process(text):
```

```
    return text.decode('rot13')

class LineOriented(WebSocket):
    def on_connect(self):
        self.read_delimiter = "\n"

    def on_read(self, line):
        self.write(process(line))

HTTPServer(LineOriented).listen(8080)
Engine.instance().start()
```

And, on the client:

```
<!DOCTYPE html>
<textarea id="editor"></textarea><br>
<input type="submit" value="Send">
<script>
    var ws = new WebSocket("ws://localhost:8080/"),
        input = document.querySelector('#editor'),
        button = document.querySelector('input');

    ws.onmessage = function(e) {
        alert("Got back: " + e.data);
    }

    button.addEventListener("click", function() {
        ws.send(input.value + "\n");
    });
</script>
```

On Python 2.x, the read delimiter will be a byte string. The WebSocket will expect to receive a byte string. However, the simple JavaScript shown above sends *unicode* strings. That simple service would fail immediately on Python 2.

To avoid the problem, be sure to use the string type you really want for your read delimiters. For the above, that's as simple as setting the read delimiter with:

```
self.read_delimiter = u"\n"
```

WebSocket Messages

In addition to the standard types of *read_delimiter*, *WebSocket* instances support the use of a special value called *EntireMessage*. When using `EntireMessage`, full messages will be sent to your *on_read* event handler, as framed by the remote end-point.

`EntireMessage` is the default *read_delimiter* for WebSocket instances, and it makes it dead simple to write simple services.

The following example implements a simple RPC system over WebSockets:

```
import json

from pants.http.server import HTTPServer
from pants.http.websocket import WebSocket, FRAME_TEXT
from pants import Engine

class RPCSocket(WebSocket):
```

```
    methods = {}

    @classmethod
    def method(cls, name):
        ''' Add a method to the RPC. '''
        def decorator(method):
            cls.methods[name] = method
            return method
        return decorator

    def json(self, **data):
        ''' Send a JSON object to the remote end-point. '''
        # JSON outputs UTF-8 encoded text by default, so use the frame
        # argument to let WebSocket know it should be sent as text to the
        # remote end-point, rather than as binary data.
        self.write(json.dumps(data), frame=FRAME_TEXT)

    def on_read(self, data):
        # Attempt to decode a JSON message.
        try:
            data = json.loads(data)
        except ValueError:
            self.json(ok=False, result="can't decode JSON")
            return

        # Lookup the desired method. Return an error if it doesn't exist.
        method = data['method']
        if not method in self.methods:
            self.json(ok=False, result="no such method")
            return

        method = self.methods[method]
        args = data.get("args", tuple())
        kwargs = data.get("kwargs", dict())
        ok = True

        # Try running the method, and capture the result. If it errors, set
        # the result to the error string and ok to False.
        try:
            result = method(*args, **kwargs)
        except Exception as ex:
            ok = False
            result = str(ex)

        self.json(ok=ok, result=result)


@RPCSocket.method("rot13")
def rot13(string):
    return string.decode("rot13")

HTTPServer(RPCSocket).listen(8080)
Engine.instance().start()
```

As you can see, it never even *uses* `read_delimiter`. The client simply sends JSON messages, with code such as:

```
my_websocket.send(JSON.stringify({method: "rot13", args: ["test"]}));
```

This behavior is completely reliable, even when the client is sending fragmented messages.

WebSocket

class pants.http.websocket.**WebSocket**(*request*, **arguments*)

An implementation of WebSockets on top of the Pants HTTP server using an API similar to that of *pants.stream.Stream*.

A *WebSocket* instance represents a WebSocket connection to a remote client. In the future, WebSocket will be modified to support acting as a client in addition to acting as a server.

When using WebSockets you write logic as you could for *Stream*, using the same *read_delimiter* and event handlers, while the WebSocket implementation handles the initial negotiation and all data framing for you.

Argument	Description
request	The *HTTPRequest* to begin negotiating a WebSocket connection over.

is_secure

Whether or not the underlying HTTP connection is secured.

allow_old_handshake

Whether or not to allow clients using the old draft-76 protocol to connect. By default, this is set to False.

Due to the primitive design of the draft-76 version of the WebSocket protocol, there is significantly reduced functionality when it is being used.

1. Binary data cannot be transmitted. All communications between the *WebSocket* instance and the remote end-point must take place using unicode strings.
2. Connections are closed immediately with no concept of close reasons. When you use *close()* on a draft-76 WebSocket, it will flush the buffer and then, once the buffer empties, close the connection immediately.
3. There are no control frames, such as the PING frames created when you invoke *ping()*.

There are other missing features as well, such as extensions and the ability to fragment long messages, but Pants does not currently provide support for those features at this time.

buffer_size

The maximum size, in bytes, of the internal buffer used for incoming data.

When buffering data it is important to ensure that inordinate amounts of memory are not used. Setting the buffer size to a sensible value can prevent coding errors or malicious use from causing your application to consume increasingly large amounts of memory. By default, a maximum of 64kb of data will be stored.

The buffer size is mainly relevant when using a string value for the *read_delimiter*. Because you cannot guarantee that the string will appear, having an upper limit on the size of the data is appropriate.

If the read delimiter is set to a number larger than the buffer size, the buffer size will be increased to accommodate the read delimiter.

When the internal buffer's size exceeds the maximum allowed, the *on_overflow_error()* callback will be invoked.

Attempting to set the buffer size to anything other than an integer or long will raise a TypeError.

close(*flush=True*, *reason=1000*, *message=None*)

Close the WebSocket connection. If flush is True, wait for any remaining data to be sent and send a close frame before closing the connection.

Argument	Default	Description
flush	`True`	*Optional.* If False, closes the connection immediately, without ensuring all buffered data is sent.
reason	`1000`	*Optional.* The reason the socket is closing, as defined at **RFC 6455#section-7.4**.
message	`None`	*Optional.* A message string to send with the reason code, rather than the default.

local_address
The address of the WebSocket on the local machine.

By default, this will be the value of `socket.getsockname` or None. It is possible for user code to override the default behaviour and set the value of the property manually. In order to return the property to its default behaviour, user code then has to delete the value. Example:

```
# default behaviour
channel.local_address = custom_value
# channel.local_address will return custom_value now
del channel.local_address
# default behaviour
```

on_close()
Placeholder. Called after the WebSocket has finished closing.

on_connect(**arguments*)
Placeholder. Called after the WebSocket has connected to a client and completed its handshake. Any additional arguments passed to the *WebSocket* instance's constructor will be passed to this method when it is invoked, making it easy to use *WebSocket* together with the URL variables captured by *pants.web.application.Application*, as shown in the following example:

```
from pants.web import Application
from pants.http import WebSocket

app = Application()
@app.route("/ws/<int:id>")
class MyConnection(WebSocket):
    def on_connect(self, id):
        pass
```

on_handshake(*request*, *headers*)
Placeholder. Called during the initial handshake, making it possible to validate the request with custom logic, such as Origin checking and other forms of authentication.

If this function returns a False value, the handshake will be stopped and an error will be sent to the client.

Argument	Description
request	The *pants.http.server.HTTPRequest* being upgraded to a WebSocket.
headers	An empty dict. Any values set here will be sent as headers when accepting (or rejecting) the connection.

on_overflow_error(*exception*)
Placeholder. Called when an internal buffer on the WebSocket has exceeded its size limit.

By default, logs the exception and closes the WebSocket.

Argument	Description
exception	The exception that was raised.

on_pong(*data*)
Placeholder. Called when a PONG control frame is received from the remote end-point in response to an

earlier ping.

When used together with the *ping()* method, on_pong may be used to measure the connection's round-trip time. See *ping()* for more information.

Argument	Description
data	Either the RTT expressed as seconds, or an arbitrary byte string that served as the PONG frame's payload.

on_read(*data*)

Placeholder. Called when data is read from the WebSocket.

Argument	Description
data	A chunk of data received from the socket. Binary data will be provided as a byte string, and text data will be provided as a unicode string.

on_write()

Placeholder. Called after the WebSocket has finished writing data.

ping(*data=None*)

Write a ping frame to the WebSocket. You may, optionally, provide a byte string of data to be used as the ping's payload. When the end-point returns a PONG, and the *on_pong()* method is called, that byte string will be provided to on_pong. Otherwise, on_pong will be called with the elapsed time.

Argument	Description
data	*Optional.* A byte string of data to be sent as the ping's payload.

read_delimiter

The magical read delimiter which determines how incoming data is buffered by the WebSocket.

As data is read from the socket, it is buffered internally by the WebSocket before being passed to the *on_read()* callback. The value of the read delimiter determines when the data is passed to the callback. Valid values are None, a string, an integer/long, a compiled regular expression, an instance of struct.Struct, an instance of netstruct.NetStruct, or the *EntireMessage* object.

When the read delimiter is the EntireMessage object, entire WebSocket messages will be passed to *on_read()* immediately once they have been received in their entirety. This is the default behavior for *WebSocket* instances.

When the read delimiter is None, data will be passed to *on_read()* immediately after it has been received.

When the read delimiter is a byte string or unicode string, data will be buffered internally until that string is encountered in the incoming data. All data up to but excluding the read delimiter is then passed to *on_read()*. The segment matching the read delimiter itself is discarded from the buffer.

Note: When using strings as your read delimiter, you must be careful to use unicode strings if you wish to send and receive strings to a remote JavaScript client.

When the read delimiter is an integer or a long, it is treated as the number of bytes to read before passing the data to *on_read()*.

When the read delimiter is an instance of struct.Struct, the Struct's size is fully buffered and the data is unpacked before the unpacked data is sent to *on_read()*. Unlike other types of read delimiters, this can result in more than one argument being sent to the *on_read()* event handler, as in the following example:

```
import struct
from pants.http import WebSocket

class Example(WebSocket):
    def on_connect(self):
        self.read_delimiter = struct.Struct("!ILH")

    def on_read(self, packet_type, length, id):
        pass
```

You must send binary data from the client when using structs as your read delimiter. If Pants receives a unicode string while a struct read delimiter is set, it will close the connection with a protocol error. This holds true for the `Netstruct` delimiters as well.

When the read delimiter is an instance of `netstruct.NetStruct`, the NetStruct's `minimum_size` is buffered and unpacked with the NetStruct, and additional data is buffered as necessary until the NetStruct can be completely unpacked. Once ready, the data will be passed to *`on_read()`*. Using Struct and NetStruct are *very* similar.

When the read delimiter is a compiled regular expression (`re.RegexObject`), there are two possible behaviors that you may switch between by setting the value of `regex_search`. If `regex_search` is True, as is the default, the delimiter's `search()` method is used and, if a match is found, the string before that match is passed to *`on_read()`*. The segment that was matched by the regular expression will be discarded.

If `regex_search` is False, the delimiter's `match()` method is used instead and, if a match is found, the match object itself will be passed to *`on_read()`*, giving you access to the capture groups. Again, the segment that was matched by the regular expression will be discarded from the buffer.

Attempting to set the read delimiter to any other value will raise a `TypeError`.

The effective use of the read delimiter can greatly simplify the implementation of certain protocols.

`remote_address`
The remote address to which the WebSocket is connected.

By default, this will be the value of `socket.getpeername` or None. It is possible for user code to override the default behaviour and set the value of the property manually. In order to return the property to its default behaviour, user code then has to delete the value. Example:

```
# default behaviour
channel.remote_address = custom_value
# channel.remote_address will return custom_value now
del channel.remote_address
# default behaviour
```

`write`(*data*, *frame=None*, *flush=False*)
Write data to the WebSocket.

Data will not be written immediately, but will be buffered internally until it can be sent without blocking the process.

Calling *`write()`* on a closed or disconnected WebSocket will raise a `RuntimeError`.

If data is a unicode string, the data will be sent to the remote end-point as text using the frame opcode for text. If data is a byte string, the data will be sent to the remote end-point as binary data using the frame opcode for binary data. If you manually specify a frame opcode, the provided data *must* be a byte string.

An appropriate header for the data will be generated by this method, using the length of the data and the frame opcode.

Arguments	Description
data	A string of data to write to the WebSocket. Unicode will be converted automatically.
frame	*Optional.* The frame opcode for this message.
flush	*Optional.* If True, flush the internal write buffer. See `pants.stream.Stream.flush()` for details.

write_file(*sfile*, *nbytes=0*, *offset=0*, *flush=False*)
Write a file to the WebSocket.

This method sends an entire file as one huge binary frame, so be careful with how you use it.

Calling `write_file()` on a closed or disconnected WebSocket will raise a `RuntimeError`.

Arguments	Description
sfile	A file object to write to the WebSocket.
nbytes	*Optional.* The number of bytes of the file to write. If 0, all bytes will be written.
offset	*Optional.* The number of bytes to offset writing by.
flush	*Optional.* If True, flush the internal write buffer. See `flush()` for details.

write_packed(**data*, ***kwargs*)
Write packed binary data to the WebSocket.

Calling `write_packed()` on a closed or disconnected WebSocket will raise a `RuntimeError`.

If the current `read_delimiter` is an instance of `struct.Struct` or `netstruct.NetStruct` the format will be read from that Struct, otherwise you will need to provide a `format`.

Argument	Description
*data	Any number of values to be passed through `struct` and written to the remote host.
format	*Optional.* A formatting string to pack the provided data with. If one isn't provided, the read delimiter will be used.
flush	*Optional.* If True, flush the internal write buffer. See `flush()` for details.

EntireMessage

pants.http.websocket.**EntireMessage**
`EntireMessage` is a unique Python object that, when set as the `read_delimiter` for a `WebSocket` instance, will cause entire messages to be passed to the `on_read()` event handler at once.

pants.web.application

`pants.web.application` implements a minimalistic framework for building websites on top of Pants.

The `Application` class features a powerful, easy to use request routing system and an API similar to that of the popular Flask project.

Note: Application does not provide out of the box support for sessions or templates, and it is not compatible with WSGI middleware as it is not implemented via WSGI.

Applications

Instances of the *Application* class are callable and act as request handlers for the *pants.http.server.HTTPServer* class. As such, to implement a server you just have to create an *HTTPServer* instance using your application.

```
from pants.http import HTTPServer
from pants.web import Application

app = Application()

HTTPServer(app).listen(8080)
```

Alternatively, you may call the Application's *run()* method, which creates an instance of HTTPServer for you and starts Pants' global *engine*.

The main features of an Application are its powerful request routing table and its output handling.

Routing

When registering new request handlers with an *Application* instance, you are required to provide a specially formatted rule. These rules allow you to capture variables from URLs on top of merely routing requests, making it easy to create attractive URLs bereft of unfriendly query strings.

Rules in their simplest form will match a static string.

```
@app.route("/")
def index(request):
    return "Index Page"

@app.route("/welcome")
def welcome(request):
    return "Hello, Programmer!"
```

Such an Application would have two pages, and not be exceptionally useful by any definition. Adding a simple variable makes things much more interesting.

```
@app.route("/welcome/<name>")
def welcome(request, name):
    return "Hello, %s!" % name
```

Variables are created using inequality signs, as demonstrated above, and allow you to capture data directly from a URL. By default, a variable accepts any character except a slash (`/`) and returns the entire captured string as an argument to your request handler.

It is possible to change this behavior by naming a *Converter* within the variable definition using the format `<converter:name>` where `converter` is the name of the converter to use. It is not case-sensitive. For example, the `int` converter:

```
@app.route("/user/<int:id>")
def user(request, id):
    return session.query(User).filter_by(id=id).first().username
```

In the above example, the `id` is automatically converted to an integer by the framework. The converter also serves to limit the URLs that will match a rule. Variables using the `int` converter will only match numbers.

Finally, you may provide default values for variables:

```
@app.route("/page/<path:slug=welcome>")
```

Default values are used if there is no string to capture for the variable in question, and are processed via the converter's *decode()* method each time the rule is matched.

When using default values, they allow you to omit the entirety of the URL following the point at which they are used. As such, if you have a rule such as `/page/<int:id=2>/other`, the URL `/page/` will match it.

Domains

The route rule strings are very similar to those used by the popular Flask framework. However, in addition to that behavior, the Application allows you to match and extract variables from the domain the page was requested from.

```
@app.route("<username>.my-site.com/blog/<int:year>/<slug>")
```

To use domains, simply place the domain before the first slash in the route rule.

Rule Variable Converters

Converters are all subclasses of *Converter* that have been registered with Pants using the *register_converter()* decorator.

A Converter has three uses:

1. Generating a regular expression snippet that will match only valid input for the variable in question.
2. Processing the captured string into useful data for the Application.
3. Encoding values into URL-friendly strings for inclusion into URLs generated via the *url_for()* method.

Converters can accept configuration information from rules using a basic format.

```
@app.route("/page/<regex('(\d{3}-\d{4})'):number>")

@app.route("/user/<id(digits=4 min=200):id>")
```

Configuration must be provided within parenthesis, with separate values separated by simple spaces. Strings may be enclosed within quotation marks if they need to contain spaces.

The values `true`, `false`, and `none` are converted to the appropriate Python values before being passed to the Converter's configuration method and it also attempts to convert values into integers or floats if possible. Use quotation marks to avoid this behavior if required.

Arguments may be passed by order or by key, and are passed to the Converter's *configure()* method from the constructor via: `self.configure(*args, **kwargs)`

Several basic converters have been included by default to make things easier.

Any

The `any` converter will allow you to match one string from a list of possible strings.

```
@app.route("/<any(call text im):action>/<int:id>")
```

Using the above rule, you can match URLs starting with `/call/`, `/text/`, or `/im/` (and followed, of course, by an integer named id).

DomainPart

DomainPart is a special converter used when matching sections of a domain name that will not match a period (`.`) but that otherwise works identically to the default String converter.

You do not have to specify the DomainPart converter. It will be used automatically in place of String for any variable capture within the domain name portion of the rule.

Float

The `float` converter will match a negation, the digits 0 through 9, and a single period. It automatically converts the captured string into a floating point number.

Argument	Default	Description
min	None	The minimum value to allow.
max	None	The maximum value to allow.

Values outside of the range defined by `min` and `max` will result in an error and *not* merely the rule not matching the URL.

Integer

The `int` (or `integer`) converter will match a negation and the digits 0 through 9, automatically converting the captured string into an integer.

Argument	Default	Description
digits	None	The exact number of digits to match with this variable.
min	None	The minimum value to allow.
max	None	The maximum value to allow.

As with the Float converter, values outside of the range defined by `min` and `max` will result in an error and *not* merely the rule not matching the URL.

Path

The `path` converter will match any character at all and merely returns the captured string. This is useful as a catch all for placing on the end of URLs.

Regex

The `regex` converter allows you to specify an arbitrary regular expression snippet for inclusion into the rule's final expression.

Argument	Default	Description
match		A regular expression snippet for inclusion into the rule's final expression.
namegen	None	The string format to use when building a URL for this variable with *`url_for()`*.

```
@app.route("/call/<regex('(\d{3}-\d{4})'):number>")
```

The above variable would match strings such as `555-1234`.

String

The `string` converter is the default converter used when none is specified, and it matches any character except for a slash (`/`), allowing it to easily capture individual URL segments.

Argument	Default	Description
min	None	The minimum length of the string to capture.
max	None	The maximum length of the string to capture.
length	None	An easy way to set both `min` and `max` at once.

Note: Setting `length` overrides any value of `min` and `max`.

Writing a Variable Converter

To create your own variable converters, you must create subclasses of *`Converter`* and register it with Pants using the decorator *`register_converter()`*.

The simplest way to use converters is as a way to store common regular expressions that you use to match segments of a URL. If, for example, you need to match basic phone numbers, you could use:

```
@app.route("/tel/<regex('(\d{3})-(\d{4})'):number>")
```

Placing the expression in the route isn't clean, however, and it can be a pain to update–particularly if you use the same expression across many different routes.

A better alternative is to use a custom converter:

```
from pants.web import Converter, register_converter

@register_converter
class Telephone(Converter):
    regex = r"(\d{3})-(\d{4})"
```

After doing that, your rule becomes as easy as `/tel/<telephone:number>`. Of course, you could stop there, and deal with the resulting tuple of two strings within your request handler.

However, the main goal of converters is to *convert* your data. Let's store our phone number in a `collections.namedtuple`. While we're at it, we'll switch to a slightly more complex regular expression that can capture area codes and extensions as well.

```
from collections import namedtuple
from pants.web import Converter, register_converter

PhoneNumber = namedtuple('PhoneNumber', ['npa','nxx','subscriber', 'ext'])

@register_converter
class Telephone(Converter):
    regex = r"(?:1[ -]*)?(?:\(? *([2-9][0-9]{2}) *\)?[ -]*)?([2-9](?:1[02-9]|[02-9][0-
↪9]))[ -]*(\d{4})(?:[ -]*e?xt?[ -]*(\d+))?"

    def decode(self, request, *values):
        return PhoneNumber(*(int(x) if x else None for x in values))
```

Now we're getting somewhere. Using our existing rule, now we can make a request for the URL `/tel/555-234-5678x115` and our request handler will receive the variable `PhoneNumber(npa=555, nxx=234, subscriber=5678, ext=115)`.

Lastly, we need a way to convert our nice `PhoneNumber` instances into something we can place in a URL, for use with the *`url_for()`* function:

```
@register_converter
class Telephone(Converter):

    ...

    def encode(self, request, value):
        out = '%03d-%03d-%04d' % (value.npa, value.nxx, value.subscriber)
        if value.ext:
            out += '-ext%d' % value.ext
        return out
```

Now, we can use `url_for('route', PhoneNumber(npa=555, nxx=234, subscriber=5678, ext=115))` and get a nice and readable `/tel/555-234-5678-ext115` back (assuming the rule for `route` is `/tel/<telephone:number>`).

Output Handling

Sending output from a request handler is as easy as returning a value from the function. Strings work well:

```
@app.route("/")
def index(request):
    return "Hello, World!"
```

The example above would result in a `200 OK` response with the headers `Content-Type: text/plain` and `Content-Length: 13`.

Response Body

If the returned string begins with `<!DOCTYPE` or `<html` it will be assumed that the `Content-Type` should be `text/html` if a content type is not provided.

If a unicode string is returned, rather than a byte string, it will be encoded automatically using the encoding specified in the `Content-Type` header. If that header is missing, or does not contain an encoding, the document will be encoded in `UTF-8` by default and the content type header will be updated.

Dictionaries, lists, and tuples will be automatically converted into JSON and the `Content-Type` header will be set to `application/json`, making it easy to send JSON to clients.

If any other object is returned, the Application will attempt to cast it into a byte string using `str(object)`. To provide custom behavior, an object may be given a `to_html` method, which will be called rather than `str()`. If `to_html` is used, the `Content-Type` will be assumed to be `text/html`.

Status and Headers

Of course, in any web application it is useful to be able to return custom status codes and HTTP headers. To do so from an Application's request handlers, simply return a tuple of `(body, status)` or `(body, status, headers)`.

If provided, `status` must be an integer or a byte string. All valid HTTP response codes may be sent simply by using their numbers.

If provided, `headers` must be either a dictionary, or a list of tuples containing key/value pairs (`[(heading, value), ...]`).

You may also use an instance of `pants.web.application.Response` rather than a simple body or tuple.

The following example returns a page with the status code `404 Not Found`:

```
@app.route("/nowhere/")
def nowhere(request):
    return "This does not exist.", 404
```

Helper Functions

`pants.web.application.`**`abort`**(*status=404*, *message=None*, *headers=None*)
Raise a `HTTPException` to display an error page.

`pants.web.application.`**`all_or_404`**(**args*)
If any of the provided arguments aren't truthy, raise a `404 Not Found` exception. This is automatically called for you if you set `auto404=True` when using the route decorator.

`pants.web.application.`**`error`**(*message=None*, *status=None*, *headers=None*, *request=None*, *debug=None*)
Return a very simple error page, defaulting to a `404 Not Found` error if no status code is supplied. Usually, you'll want to call *`abort()`* in your code, rather than error(). Usage:

```
return error(404)
return error("Some message.", 404)
return error("Blah blah blah.", 403, {'Some-Header': 'Fish'})
```

`pants.web.application.`**`redirect`**(*url*, *status=302*, *request=None*)
Construct a `302 Found` response to instruct the client's browser to redirect its request to a different URL. Other codes may be returned by specifying a status.

Argument	Default	Description
url		The URL to redirect the client's browser to.
status	`302`	*Optional.* The status code to send with the response.

`pants.web.application.`**`register_converter`**(*name=None*, *klass=None*)
Register a converter with the given name. If a name is not provided, the class name will be converted to lowercase and used instead.

`pants.web.application.`**`url_for`**(*name*, **values*, ***kw_values*)
Generates a URL for the route with the given name. You may give either an absolute name for the route or use a period to match names relative to the current route. Multiple periods may be used to traverse up the name tree.

Passed arguments will be used to construct the URL. Any unknown keyword arguments will be appended to the URL as query arguments. Additionally, there are several special keyword arguments to customize `url_for`'s behavior.

Argument	Default	Description
_anchor	None	*Optional.* An anchor string to be appended to the URL.
_doseq	True	*Optional.* The value to pass to `urllib.urlencode()`'s `doseq` parameter for building the query string.
_external	False	*Optional.* Whether or not a URL is meant for external use. External URLs never have their host portion removed.
_scheme	None	*Optional.* The scheme of the link to generate. By default, this is set to the scheme of the current request.

Application

class pants.web.application.**Application**(*name=None*, *debug=False*, *fix_end_slash=False*)

The Application class builds upon the *Module* class and acts as a request handler for the *HTTPServer*, with request routing, error handling, and a degree of convenience that makes sending output easier.

Instances of Application are callable, and should be used as a HTTPServer's request handler.

Argument	Description
debug	*Optional.* If this is set to True, the automatically generated `500 Internal Server Error` pages will display additional debugging information.

run(*address=None*, *ssl_options=None*, *engine=None*)

This function exists for convenience, and when called creates a *HTTPServer* instance with its request handler set to this application instance, calls *listen()* on that HTTPServer, and finally, starts the Pants engine to process requests.

Argument	Description
address	*Optional.* The address to listen on. If this isn't specified, it will default to `('', 80)`.
ssl_options	*Optional.* A dictionary of SSL options for the server. See *pants.server.Server.startSSL()* for more information.
engine	*Optional.* The *pants.engine.Engine* instance to use.

Module

class pants.web.application.**Module**(*name=None*)

A Module is, essentially, a group of rules for an Application. Rules grouped into a Module can be created without any access to the final Application instance, making it simple to split a website into multiple Python modules to be imported in the module that creates and runs the application.

add(*rule*, *module*)

Add a Module to this Module under the given rule. All rules within the sub-module will be accessible to this Module, with their rules prefixed by the rule provided here.

For example:

```
module_one = Module()

@module_one.route("/fish")
def fish(request):
    return "This is fish."

module_two = Module()

module_two.add("/pie", module_one)
```

Given that code, the request handler `fish` would be available from the Module `module_two` with the rules `/pie/fish`.

basic_route(*rule*, *name=None*, *methods=('GET', 'HEAD')*, *headers=None*, *content_type=None*, *func=None*)

The basic_route decorator registers a route with the Module without holding your hand about it.

It functions similarly to the *Module.route()* decorator, but it doesn't wrap the function with any argument processing code. Instead, the function is given only the request object, and through it access to the regular expression match.

Example Usage:

```
@app.basic_route("/char/<char>")
def my_route(request):
    char, = request.match.groups()
    return "The character is %s!" % char
```

That is essentially equivalent to:

```
@app.route("/char/<char>")
def my_route(request, char):
    return "The character is %s!" % char
```

Note: Output is still handled the way it is with a normal route, so you can return strings and dictionaries as usual.

Argument	Description
rule	The route rule to match for a request to go to the decorated function. See *Module.route()* for more information.
name	*Optional.* The name of the decorated function, for use with the *url_for()* helper function.
methods	*Optional.* A list of HTTP methods to allow for this request handler. By default, only `GET` and `HEAD` requests are allowed, and all others will result in a `405 Method Not Allowed` error.
headers	*Optional.* A dictionary of HTTP headers to always send with the response from this request handler. Any headers set within the request handler will override these headers.
content_type	*Optional.* The HTTP Content-Type header to send with the response from this request handler. A Content-Type header set within the request handler will override this.
func	*Optional.* The function for this view. Specifying the function bypasses the usual decorator-like behavior of this function.

request_finished(*func*)

Register a method to be executed immediately after the request handler and before the output is processed and send to the client.

This can be used to transform the output of request handlers.

Note: These hooks are not run if there is no matching rule for a request, if there is an exception while running the request handler, or if the request is not set to have its output processed by the Application by setting `request.auto_finish` to `False`.

request_started(*func*)

Register a method to be executed immediately after a request has been successfully routed and before the request handler is executed.

Note: Hooks, including `request_started`, are not executed if there is no matching rule to handle the request.

This can be used for the initialization of sessions, a database connection, or other details. However, it is not always the best choice. If you wish to modify *all* requests, or manipulate the URL before routing occurs,

you should wrap the Application in another method, rather than using a `request_started` hook. As an example of the difference:

```
from pants.web import Application
from pants.http import HTTPServer
from pants import Engine

from my_site import sessions, module

app = Application()

# The Hook
@app.request_started
def handle(request):
    logging.info('Request matched route: %s' % request.route_name)

# The Wrapper
def wrapper(request):
    request.session = sessions.get(request.get_secure_cookie('session_id'))
    app(request)

# Add rules from another module.
app.add('/', module)

HTTPServer(wrapper).listen()
Engine.instance().start()
```

request_teardown (*func*)

Register a method to be executed after the output of a request handler has been processed and has begun being transmitted to the client. At this point, the request is not going to be used again and can be cleaned up.

Note: These hooks will always run if there was a matching rule, even if the request handler or other hooks have exceptions, to prevent any potential memory leaks from requests that aren't torn down properly.

---

route (*rule*, *name=None*, *methods=('GET', 'HEAD')*, *auto404=False*, *headers=None*, *content_type=None*, *func=None*)

The route decorator is used to register a new route with the Module instance. Example:

```
@app.route("/")
def hello_world(request):
    return "Hiya, Everyone!"
```

See also:

See *Routing* for more information on writing rules.

Argument	Description
rule	The route rule to be matched for the decorated function to be used for handling a request.
name	*Optional.* The name of the decorated function, for use with the *url_for()* helper function.
methods	*Optional.* A list of HTTP methods to allow for this request handler. By default, only `GET` and `HEAD` requests are allowed, and all others will result in a `405 Method Not Allowed` error.
auto404	*Optional.* If this is set to True, all response handler arguments will be checked for truthiness (True, non-empty strings, etc.) and, if any fail, a `404 Not Found` page will be rendered automatically.
headers	*Optional.* A dictionary of HTTP headers to always send with the response from this request handler. Any headers set within the request handler will override these headers.
content_type	*Optional.* The HTTP Content-Type header to send with the response from this request handler. A Content-Type header set within the request handler will override this.
func	*Optional.* The function for this view. Specifying the function bypasses the usual decorator-like behavior of this function.

Converter

class pants.web.application.**Converter**(*options*, *default*)

The Converter class is the base class for all the different value converters usable in routing rules.

default

A string provided with the variable declaration to be used as a default value if no value is provided by the client.

This value will also be placed in urls generated via the method *url_for()* if no other value is provided.

configure()

The method receives configuration data parsed from the rule creating this Converter instance as positional and keyword arguments.

You must build a regular expression for matching acceptable input within this function, and save it as the instance's `regex` attribute. You may use more than one capture group.

decode(*request*, **values*)

This method receives captured strings from URLs and must process the strings and return variables usable within request handlers.

If the converter's regular expression has multiple capture groups, it will receive multiple arguments.

Note: Use *abort()* or raise an *HTTPException* from this method if you wish to display an error page. Any other uncaught exceptions will result in a `400 Bad Request` page.

---

encode(*request*, *value*)

This method encodes a value into a URL-friendly string for inclusion into URLs generated with *url_for()*.

Exceptions

class pants.web.application.**HTTPException**(*status=404*, *message=None*, *headers=None*)

Raising an instance of HTTPException will cause the Application to render an error page out to the client with the given HTTP status code, message, and any provided headers.

This is, generally, preferable to allowing an exception of a different type to bubble up to the Application, which would result in a `500 Internal Server Error` page.

The *`abort()`* helper function makes it easy to raise instances of this exception.

Argument	Description
status	*Optional.* The HTTP status code to generate an error page for. If this isn't specified, a `404 Not Found` page will be generated.
message	*Optional.* A text message to display on the error page.
headers	*Optional.* A dict of extra HTTP headers to return with the rendered page.

class `pants.web.application.`**`HTTPTransparentRedirect`**(*url*)

Raising an instance of HTTPTransparentRedirect will cause the Application to silently redirect a request to a new URL.

`pants.web.fileserver`

`pants.web.fileserver` implements a basic static file server for use with a *`HTTPServer`* or *`Application`*. It makes use of the appropriate HTTP headers and the `sendfile` system call, as well as the `X-Sendfile` header to improve transfer performance.

Serving Static Files

The `pants.web.fileserver` module can be invoked directly using the *-m* switch of the interpreter to serve files in a similar way to the standard library's `SimpleHTTPServer`. However, it performs much more efficiently than `SimpleHTTPServer` for this task.

```
$ python -m pants.web.fileserver
```

When doing this, you may use additional arguments to specify which address the server should bind to, as well as which filenames should serve as directory indices. By default, only `index.html` and `index.htm` are served as indices.

`FileServer`

class `pants.web.fileserver.`**`FileServer`**(*path*, *blacklist=(<_sre.SRE_Pattern object>,)*, *defaults=('index.html', 'index.htm')*)

The FileServer is a request handling class that, as it sounds, serves files to the client using *`pants.http.server.HTTPRequest.send_file()`*. As such, it supports caching headers, as well as `X-Sendfile` if the *`HTTPServer`* instance is configured to use the Sendfile header. FileServer is also able to take advantage of the `sendfile` system call to improve performance when `X-Sendfile` is not in use.

Argument	Default	Description
path		The path to serve.
blacklist	`.py` and `.pyc` files	*Optional.* A list of regular expressions to test filenames against. If a given file matches any of the provided patterns, it will not be downloadable and instead return a `403 Unauthorized` error.
default	`index.html`, `index.htm`	*Optional.* A list of default files to be displayed rather than a directory listing if they exist.

Using it is simple. It only requires a single argument: the path to serve files from. You can also supply a list of default files to check to serve rather than a file listing.

When used with an Application, the FileServer is not created in the usual way with the route decorator, but rather with a method of the FileServer itself. Example:

```
FileServer("/tmp/path").attach(app)
```

If you wish to listen on a path other than `/static/`, you can also use that when attaching:

```
FileServer("/tmp/path").attach(app, "/files/")
```

attach(*app*, *path='/static/'*)

Attach this FileServer to an *Application*, bypassing the usual route decorator to ensure the rule is configured as FileServer expects.

Argument	Default	Description
app		The `Application` instance to attach to.
rule	`'/static/'`	*Optional.* The path to serve requests from.

pants.web.wsgi

`pants.web.wsgi` implements a WSGI compatibility class that lets you run WSGI applications using the Pants *HTTPServer*.

Currently, this module uses the **PEP 333** standard. Future releases will add support for **PEP 3333**, as well as the ability to host a Pants *Application* from a standard WSGI server.

WSGIConnector

class `pants.web.wsgi.`**WSGIConnector**(*application*, *debug=False*)

This class functions as a request handler for the Pants *HTTPServer* that wraps WSGI applications to allow them to work correctly.

Class instances are callable, and when called with a *HTTPRequest* instance, they construct a WSGI environment and invoke the application.

```
from pants import Engine
from pants.http import HTTPServer
from pants.web import WSGIConnector

def hello_app(environ, start_response):
    start_response("200 OK", {"Content-Type": "text/plain"})
    return ["Hello, World!"]

connector = WSGIConnector(hello_app)
HTTPServer(connector).listen()
Engine.instance().start()
```

`WSGIConnector` supports sending responses with `Transfer-Encoding: chunked` and will do so automatically when the WSGI application's response does not contain information about the response's length.

Argument	Description
application	The WSGI application that will handle incoming requests.
debug	*Optional.* Whether or not to display tracebacks and additional debugging information for a request within `500 Internal Server Error` pages.

attach(*application*, *rule*, *methods=('HEAD', 'GET', 'POST', 'PUT')*)

Attach the WSGIConnector to an instance of *Application* at the given *route*.

You may use route variables to strip information out of a URL. In the event that variables exist, they will be made available within the WSGI environment under the key wsgiorg.routing_args

Warning: When using WSGIConnector within an Application, WSGIConnector expects the final variable in the rule to capture the remainder of the URL, and it treats the last variable as containing the value for the `PATH_INFO` variable in the WSGI environment. This method adds such a variable automatically. However, if you add the WSGIConnector manually you will have to be prepared.

Argument	Description
application	The `Application` to attach to.
rule	The path to serve requests from.
methods	*Optional.* The HTTP methods to accept.

Contributions

pants.contrib.irc

BaseIRC

class pants.contrib.irc.**BaseIRC**(*encoding='utf-8'*, ***kwargs*)

The IRC protocol, implemented over a Pants *Stream*.

The goal with this is to create a lightweight IRC class that can serve as either a server or a client. As such, it doesn't implement a lot of logic in favor of providing a robust base.

The BaseIRC class can receive and send IRC commands, and automatically respond to certain commands such as PING.

This class extends *Stream*, and as such has the same *connect()* and `listen()` functions.

irc_close()

Placeholder.

This method is called whenever the IRC instance becomes disconnected from the remote client or server.

irc_command(*command*, *args*, *nick*, *user*, *host*)

Placeholder.

This method is called whenever a command is received from the other side and successfully parsed as an IRC command.

Argument	Description
command	The received command.
args	A list of the arguments following the command.
nick	The nick of the user that sent the command, if applicable, or an empty string.
user	The username of the user that sent the command, if applicable, or an empty string.
host	The host of the user that sent the command, the host of the server that sent the command, or an empty string if no host was supplied.

irc_connect()
Placeholder.

This method is called when the IRC instance has successfully connected to the remote client or server.

message(*destination*, *message*, *_ctcpQuote=True*, *_prefix=None*)
Send a message to the given nick or channel.

Argument	Default	Description
destination		The nick or channel to send the message to.
message		The text of the message to be sent.
_ctcpQuote	True	*Optional.* If True, the message text will be quoted for CTCP before being sent.
_prefix	None	*Optional.* A string that, if provided, will be prepended to the command string before it's sent to the server.

notice(*destination*, *message*, *_ctcpQuote=True*, *_prefix=None*)
Send a NOTICE to the specified destination.

Argument	Default	Description
destination		The nick or channel to send the NOTICE to.
message		The text of the NOTICE to be sent.
_ctcpQuote	True	*Optional.* If True, the message text will be quoted for CTCP before being sent.
_prefix	None	*Optional.* A string that, if provided, will be prepended to the command string before it's sent to the server.

quit(*reason=None*, *_prefix=None*)
Send a QUIT message, with an optional reason.

Argument	Default	Description
reason	None	*Optional.* The reason for quitting that will be displayed to other users.
_prefix	None	*Optional.* A string that, if provided, will be prepended to the command string before it's sent to the server.

send_command(*command*, **args*, ***kwargs*)
Send a command to the remote endpoint.

Argument	Default	Description
command		The command to send.
*args		*Optional.* A list of arguments to send with the command.
_prefix	None	*Optional.* A string that, if provided, will be prepended to the command string before it's sent to the server.

Channel

class pants.contrib.irc.**Channel**(*name*)
An IRC channel's representation, for keeping track of users and the topic and stuff.

IRCClient

class pants.contrib.irc.**IRCClient**(*encoding='utf-8'*, ***kwargs*)
An IRC client, written in Pants, based on *BaseIRC*.

This implements rather more logic, and keeps track of what server it's connected to, its nick, and what channels it's in.

channel(*name*)
Retrieve a Channel object for the channel name, or None if we're not in that channel.

connect(*server=None*, *port=None*)
Connect to the server.

Argument	Description
server	The host to connect to.
port	The port to connect to on the remote server.

irc_ctcp(*nick*, *message*, *user*, *host*)
Placeholder.

This method is called when the bot receives a CTCP message, which could, in theory, be anywhere in a PRIVMSG... annoyingly enough.

Argument	Description
nick	The nick of the user that sent the CTCP message, or an empty string if no nick is available.
message	The full CTCP message.
user	The username of the user that sent the CTCP message, or an empty string if no username is available.
host	The host of the user that sent the CTCP message, or an empty string if no host is available.

irc_join(*channel*, *nick*, *user*, *host*)
Placeholder.

This method is called when a user enters a channel. That also means that this function is called whenever this IRC client successfully joins a channel.

Argument	Description
channel	The channel a user has joined.
nick	The nick of the user that joined the channel.
user	The username of the user that joined the channel.
host	The host of the user that joined the channel.

irc_message_channel(*channel*, *message*, *nick*, *user*, *host*)
Placeholder.

This method is called when the client receives a message from a channel.

Argument	Description
channel	The channel the message was received in.
message	The text of the message.
nick	The nick of the user that sent the message.
user	The username of the user that sent the message.
host	The host of the user that sent the message.

irc_message_private(*nick*, *message*, *user*, *host*)
Placeholder.

This method is called when the client receives a message from a user.

Argument	Description
nick	The nick of the user that sent the message.
message	The text of the message.
user	The username of the user that sent the message.
host	The host of the user that sent the message.

irc_nick_changed(*nick*)
Placeholder.

This method is called when the client's nick on the network is changed for any reason.

Argument	Description
nick	The client's new nick.

irc_part(*channel*, *reason*, *nick*, *user*, *host*)
Placeholder.

This method is called when a leaves enters a channel. That also means that this function is called whenever this IRC client leaves a channel.

Argument	Description
channel	The channel that the user has left.
reason	The provided reason message, or an empty string if there is no message.
nick	The nick of the user that left the channel.
user	The username of the user that left the channel.
host	The host of the user that left the channel.

irc_topic_changed(*channel*, *topic*)
Placeholder.

This method is called when the topic of a channel changes.

Argument	Description
channel	The channel which had its topic changed.
topic	The channel's new topic.

join(*channel*)
Join the specified channel.

Argument	Description
channel	The name of the channel to join.

nick
This instance's current nickname on the server it's connected to, or the nickname it will attempt to acquire when connecting.

part(*channel*, *reason=None*, *force=False*)
Leave the specified channel.

Argument	Default	Description
channel		The channel to leave.
reason	None	*Optional.* The reason why.
force	False	*Optional.* Don't ensure the client is actually *in* the named channel before sending `PART`.

port
The port this instance is connected to on the remote server, or the port it will attempt to connect to.

realname
The real name this instance will report to the server when connecting.

server
The server this instance is connected to, or will attempt to connect to.

user
The user name this instance will report to the server when connecting.

Helper Functions

`pants.contrib.irc.`**`ctcpQuote`**(*message*)
Low-level quote a message, adhering to the CTCP guidelines.

`pants.contrib.irc.`**`ctcpUnquote`**(*message*)
Low-level unquote a message, adhering to the CTCP guidelines.

pants.contrib.qt

Installation

`pants.contrib.qt.`**`install`**(*app=None*, *timeout=0.02*, *engine=None*)
Creates a `QTimer` instance that will be triggered continuously to call *`Engine.poll()`*, ensuring that Pants remains responsive.

Argument	Default	Description
app	None	*Optional.* The `QCoreApplication` to attach to. If no application is provided, it will attempt to find an existing application in memory, or, failing that, create a new application instance.
timeout	`0.02`	*Optional.* The maximum time to wait, in seconds, before running *`Engine.poll()`*.
engine		*Optional.* The *`pants.engine.Engine`* instance to use.

pants.contrib.telnet

Constants

Telnet Commands

Constant	Value
IAC	`'\xFF'`
DONT	`'\xFE'`
DO	`'\xFD'`
WONT	`'\xFC'`
WILL	`'\xFB'`
SB	`'\xFA'`
SE	`'\xF0'`

TelnetConnection

class `pants.contrib.telnet.`**`TelnetConnection`**(***kwargs*)

A basic implementation of a Telnet connection.

A TelnetConnection object is capable of identifying and extracting Telnet command sequences from incoming data. Upon identifying a Telnet command, option or subnegotiation, the connection will call a relevant placeholder method. This class should be subclassed to provide functionality for individual commands and options.

`close`(*flush=True*)

Close the channel.

`on_close`()

Placeholder. Called after the channel has finished closing.

`on_command`(*command*)

Placeholder. Called when the connection receives a telnet command, such as AYT (Are You There).

Argument	Description
command	The byte representing the telnet command.

`on_connect`()

Placeholder. Called after the channel has connected to a remote socket.

`on_option`(*command*, *option*)

Placeholder. Called when the connection receives a telnet option negotiation sequence, such as IAC WILL ECHO.

Argument	Description
command	The byte representing the telnet command.
option	The byte representing the telnet option being negotiated.

`on_read`(*data*)

Placeholder. Called when data is read from the channel.

Argument	Description
data	A chunk of data received from the socket.

`on_subnegotiation`(*option*, *data*)

Placeholder. Called when the connection receives a subnegotiation sequence.

Argument	Description
option	The byte representing the telnet option for which subnegotiation data has been received.
data	The received data.

`on_write`()

Placeholder. Called after the channel has finished writing data.

write(*data*, *flush=False*)
Write data to the channel.

Data will not be written immediately, but will be buffered internally until it can be sent without blocking the process.

Calling *write()* on a closed or disconnected channel will raise a RuntimeError.

Arguments	Description
data	A string of data to write to the channel.
flush	*Optional.* If True, flush the internal write buffer. See *flush()* for details.

write_file(*sfile*, *nbytes=0*, *offset=0*, *flush=False*)
Write a file to the channel.

The file will not be written immediately, but will be buffered internally until it can be sent without blocking the process.

Calling *write_file()* on a closed or disconnected channel will raise a RuntimeError.

Arguments	Description
sfile	A file object to write to the channel.
nbytes	*Optional.* The number of bytes of the file to write. If 0, all bytes will be written.
offset	*Optional.* The number of bytes to offset writing by.
flush	*Optional.* If True, flush the internal write buffer. See *flush()* for details.

TelnetServer

class pants.contrib.telnet.**TelnetServer**(*ConnectionClass=None*, ***kwargs*)
A basic implementation of a Telnet server.

Python Module Index

p

Index

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

W