Let's read code: python-requests library

Susan Tan
Cisco in San Francisco
Twitter: @ArcTanSusan

PyCon
May 30, 2016
Portland, OR

First time you *git clone* and open a new repo

“Indeed, the ratio of time spent reading versus writing is well over 10 to 1. We are constantly reading old code as part of the effort to write new code. ...[Therefore,] making it easy to read makes it easier to write.”

Robert C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship

This is a talk about

1. how to ***actively*** read a new Python codebase
2. reading thru the python-requests codebase

Step 0: Prepare Your Editor

Set up your editor to

- jump into any method or class definition
- search files by keywords
- get call hierarchy of any given method or class

Note: I'll be using Sublime Text with python-requests library.

Step 1: Git clone and open the repo

```
$ git clone https://github.com/
kennethreitz/requests
$ cd requests
$ subl requests
```

Step 2: Set up local dev environment to get into mindset of a contributor

Development Dependencies

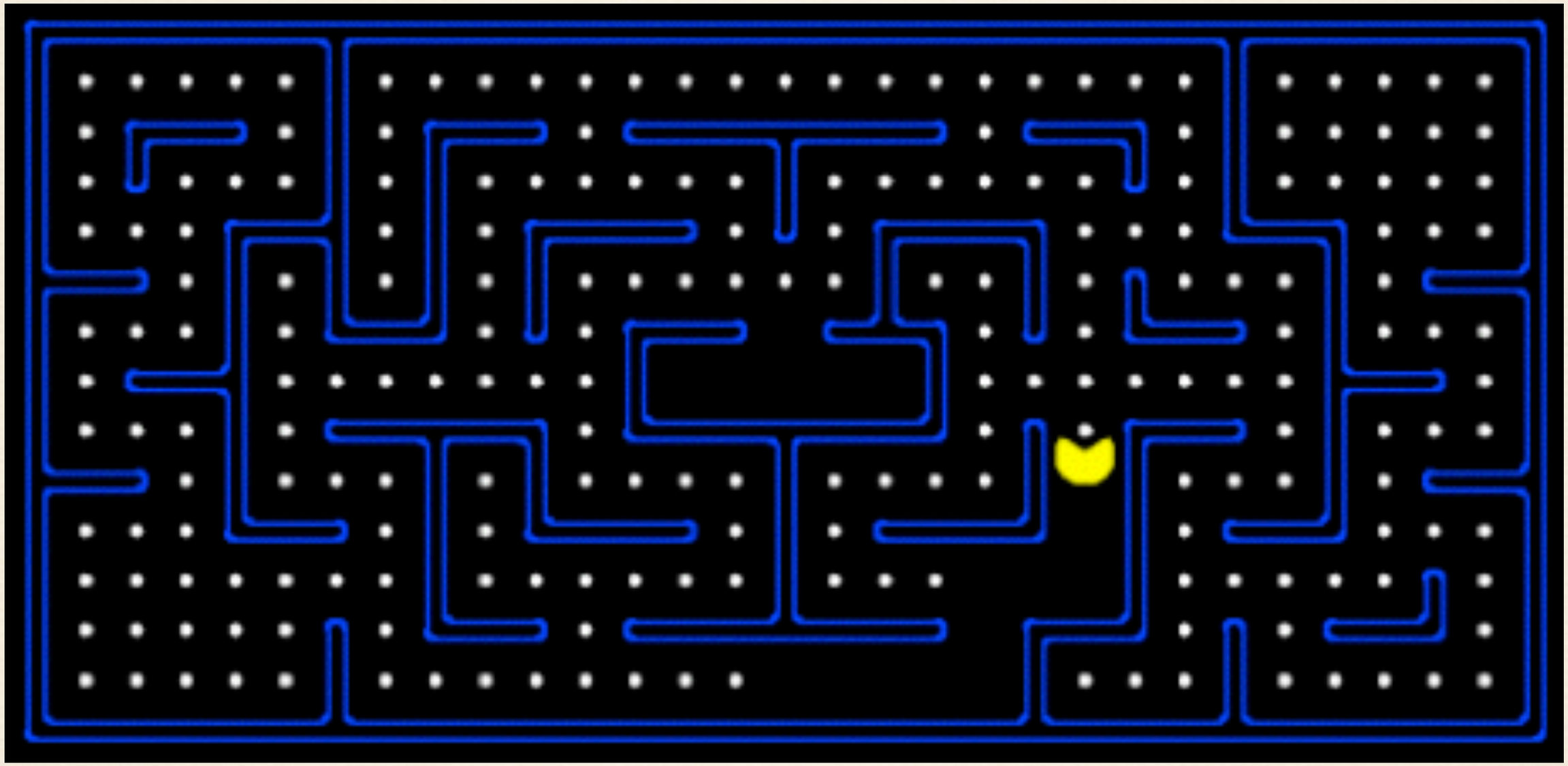
You'll need to install py.test in order to run the Requests' test suite:

```
$ pip install -r requirements.txt
$ py.test
platform darwin -- Python 2.7.3 -- pytest-2.3.4
collected 25 items

test_requests.py .........................
25 passed in 3.50 seconds
```

Source: http://docs.python-requests.org/en/latest/dev/todo/#development-dependencies

python test_requests.py works too.
requests is on permanent feature freeze.

Once you've set up your editor & local dev environment…



What it's like to read a large codebase

Goal for today — Figure out how this code snippet works

```
>>> r = requests.get('https://api.github.com/
user', auth=('user', 'pass'))
>>> r.status_code
200
>>> r.headers['content-type']
'application/json; charset=utf8'
>>> r.encoding
'utf-8'
>>> r.text
u'{"type":"User"...'
>>> r.json()
{u'private_gists': 419, u'total_private_repos':
77}
```

Step 3: Look at unit tests

- Over 1,600 lines of code in **test_requests.py**. Where to look first?
- Use git grep or keyword search for “requests.get”

git grep requests.get test_requests.py

```
$ git grep requests.get tests/test_requests.py
test_requests.py:95:        requests.get
test_requests.py:103:
requests.get('hiwpefhipowhefopw')
test_requests.py:105:           requests.get('localhost:3128')
test_requests.py:107:
……

$ git grep requests.get tests/test_requests.py | wc -l
47
```

Let’s look at *one* unit test

```
def test_DIGEST_HTTP_200_OK_GET(self, httpbin):

    auth = HTTPDigestAuth('user', 'pass')
    url = httpbin('digest-auth', 'auth',
'user', 'pass')

    r = requests.get(url, auth=auth)
    assert r.status_code == 200

    r = requests.get(url)
    assert r.status_code == 401

    s = requests.session()
    s.auth = HTTPDigestAuth('user', 'pass')
    r = s.get(url)
    assert r.status_code == 200
```

Looks like test setup happens here

test_requests.py

```
def test_DIGEST_HTTP_200_OK_GET(self, httpbin):

    auth = HTTPDigestAuth('user', 'pass')
    url = httpbin('digest-auth', 'auth', 'user',
'pass')

    r = requests.get(url, auth=auth)
    assert r.status_code == 200

    r = requests.get(url)
    assert r.status_code == 401

    s = requests.session()
    s.auth = HTTPDigestAuth('user', 'pass')
    r = s.get(url)
    assert r.status_code == 200
```

test_requests.py

```
def test_DIGEST_HTTP_200_OK_GET(self, httpbin):

    auth = HTTPDigestAuth('user', 'pass')
    url = httpbin('digest-auth', 'auth', 'user',
'pass')

    r = requests.get(url, auth=auth)
    assert r.status_code == 200

    r = requests.get(url)
    assert r.status_code == 401

    s = requests.session()
    s.auth = HTTPDigestAuth('user', 'pass')
    r = s.get(url)
    assert r.status_code == 200
```

What's a session?

test_requests.py

```
def test_DIGEST_HTTP_200_OK_GET(self, httpbin):

    auth = HTTPDigestAuth('user', 'pass')
    url = httpbin('digest-auth', 'auth', 'user',
'pass')
```

Let's look at class definition

```
    r = requests.get(url, auth=auth)
    assert r.status_code == 200

    r = requests.get(url)
    assert r.status_code == 401

    s = requests.session()
    s.auth = HTTPDigestAuth('user', 'pass')
    r = s.get(url)
    assert r.status_code == 200
```

test_requests.py

What is the HTTPDigestAuth class?

```
class HTTPDigestAuth(AuthBase):
    """Attaches HTTP Digest Authentication to the given Request
object."""
    def __init__(self, username, password):
        self.username = username
        self.password = password
        # Keep state in per-thread local storage
        self._thread_local = threading.local()

    def init_per_thread_state(self):
        # Ensure state is initialized just once per-thread
            …
        …

    def build_digest_header(self, method, url):
        …

    def handle_redirect(self, r, **kwargs):
        …

    def handle_401(self, r, **kwargs):
        …

    def __call__(self, r):
        …
```

auth.py

📖 requests has pretty good docs 📖

Digest Authentication

Another very popular form of HTTP Authentication is Digest Authentication, and Requests supports this out of the box as well:

```
>>> from requests.auth import HTTPDigestAuth
>>> url = 'http://httpbin.org/digest-auth/auth/user/pass'
>>> requests.get(url, auth=HTTPDigestAuth('user', 'pass'))
<Response [200]>
```

Source: http://docs.python-requests.org/en/latest/user/authentication/#digest-authentication

```
def test_DIGEST_HTTP_200_OK_GET(self, httpbin):

    auth = HTTPDigestAuth('user', 'pass')
    url = httpbin('digest-auth', 'auth',
'user', 'pass')
```

Let's look at method definition

```
    r = requests.get(url, auth=auth)
    assert r.status_code == 200

    r = requests.get(url)
    assert r.status_code == 401

    s = requests.session()
    s.auth = HTTPDigestAuth('user', 'pass')
    r = s.get(url)
    assert r.status_code == 200
```

test_requests.py

```
def test_DIGEST_HTTP_200_OK_GET(self, httpbin):

    auth = HTTPDigestAuth('user', 'pass')
    url = httpbin('digest-auth', 'auth',
'user', 'pass')
```

This is httpbin() method in conftest.py:

```
def prepare_url(value):
    httpbin_url = value.url.rstrip('/') + '/'

    def inner(*suffix):
        return urljoin(httpbin_url, '/'.join(suffix))
    return inner

@pytest.fixture
def httpbin(httpbin):
    return prepare_url(httpbin)
```


Im confus

I'm still confused by what *httpbin()* method is doing.

Next steps:

- **look up "httpbin" in official request docs.**
- **If that fails, then use a debugger.**

We'll do both.

Requests

Requests is an elegant and simple HTTP library for Python, built for human beings. You are currently looking at the documentation of the development release.

Buy Requests Pro

Get Updates

Receive updates on new releases and upcoming projects.

Subscribe to Newsletter

Search

From here you can search these documents. Enter your search words into the box below and click "search". Note that the search function will automatically search for all of the words. Pages containing fewer words won't appear in the result list.

Search Results

Search finished, found 5 page(s) matching the search query.

- **Advanced Usage**
  ...of the main Requests API. Let's persist some cookies across requests:: s = requests.Session() s.get('http://httpbin.org/cookies/set/sessioncookie/123456789') r = s.get("http://httpbin.org/cookies") print(r.text) # '...
- **Authentication**
  ...Requests supports this out of the box as well:: >>> from requests.auth import HTTPDigestAuth >>> url = 'http://httpbin.org/digest-auth/auth/user/pass' >>> requests.get(url, auth=HTTPDigestAuth('user', 'pass')) <Response [20...
- **Community Updates**
  .. _updates: Community Updates ========== If you'd like to stay up to date on the community and development of Requests, there are several options: GitHub ------ The best way to track the development of Requests is through `the...
- **Developer Interface**
  ...on() # formerly, session took parameters s.auth = auth s.headers.update(headers) r = s.get('http://httpbin.org/headers') * All request

Source: https://github.com/Runscope/httpbin

README.rst

httpbin(1): HTTP Request & Response Service

Freely hosted in HTTP, HTTPS & EU flavors by Runscope

Deploy to Heroku

build passing

ENDPOINTS

Endpoint	Description
/	This page.
/ip	Returns Origin IP.
/user-agent	Returns user-agent.
/headers	Returns header dict.
/get	Returns GET data.
/post	Returns POST data.
/patch	Returns PATCH data.
/put	Returns PUT data.
/delete	Returns DELETE data
/gzip	Returns gzip-encoded data.
/deflate	Returns deflate-encoded data.
/status/:code	Returns given HTTP Status code.
/response-headers	Returns given response headers.
/redirect/:n	302 Redirects *n* times.
/redirect-to?url=foo	302 Redirects to the *foo* URL.
/relative-redirect/:n	302 Relative redirects *n* times.
/cookies	Returns cookie data.

Next Step: Let's try out this http://httpbin.org/

http://httpbin.org/cookies


```
{
  - cookies: {
        _ga: "GA1.2.165241120.1453277938"
    }
}
```


http://httpbin.org/get


```
{
    args: { },
  - headers: {
        Accept: "text/html,application/xhtml+xml,a
        Accept-Encoding: "gzip, deflate, sdch",
        Accept-Language: "en-US,en;q=0.8",
        Cookie: "_ga=GA1.2.165241120.1453277938",
        Dnt: "1",
        Host: "httpbin.org",
        Upgrade-Insecure-Requests: "1",
        User-Agent: "Mozilla/5.0 (Macintosh; Intel
        Chrome/47.0.2526.106 Safari/537.36"
    },
    origin: "50.148.141.36",
    url: "http://httpbin.org/get"
}
```

httpbin's /post/ endpoint is useful for testing requests library

```
In [1]: import requests

In [2]: resp = requests.post('http://httpbin.org/post',
data={'name':'Susan'})

In [3]: resp.json()
Out[3]:
{u'args': {},
 u'data': u'',
 u'files': {},
 u'form': {u'name': u'Susan'},
 u'headers': {u'Accept': u'*/*',
  u'Accept-Encoding': u'gzip, deflate',
  u'Content-Length': u'10',
  u'Content-Type': u'application/x-www-form-urlencoded',
  u'Host': u'httpbin.org',
  u'User-Agent': u'python-requests/2.9.1'},
 u'json': None,
 u'origin': u'50.148.141.36',
 u'url': u'http://httpbin.org/post'}
```

httpbin endpoints and requests unit test

- **httpbin is everywhere in unit tests in the requests repo every time a request is made.**

- **This is a BIG step forward in our understanding of unit tests in this repo.**

Use pdbpp debugger

```
import pdb
pdb.set_trace()
```

Let’s inspect variable “url” in that previous unit test.

“pdbpp is a million times better than ipdb” —a co-worker

```
def test_DIGEST_HTTP_200_OK_GET(self, httpbin):

    auth = HTTPDigestAuth('user', 'pass')
    url = httpbin('digest-auth', 'auth',
'user', 'pass')


    r = requests.get(url, auth=auth)
    import pdb;pdb.set_trace()
    assert r.status_code == 200

    r = requests.get(url)
    assert r.status_code == 401

    s = requests.session()
    s.auth = HTTPDigestAuth('user', 'pass')
    r = s.get(url)
    assert r.status_code == 200
```

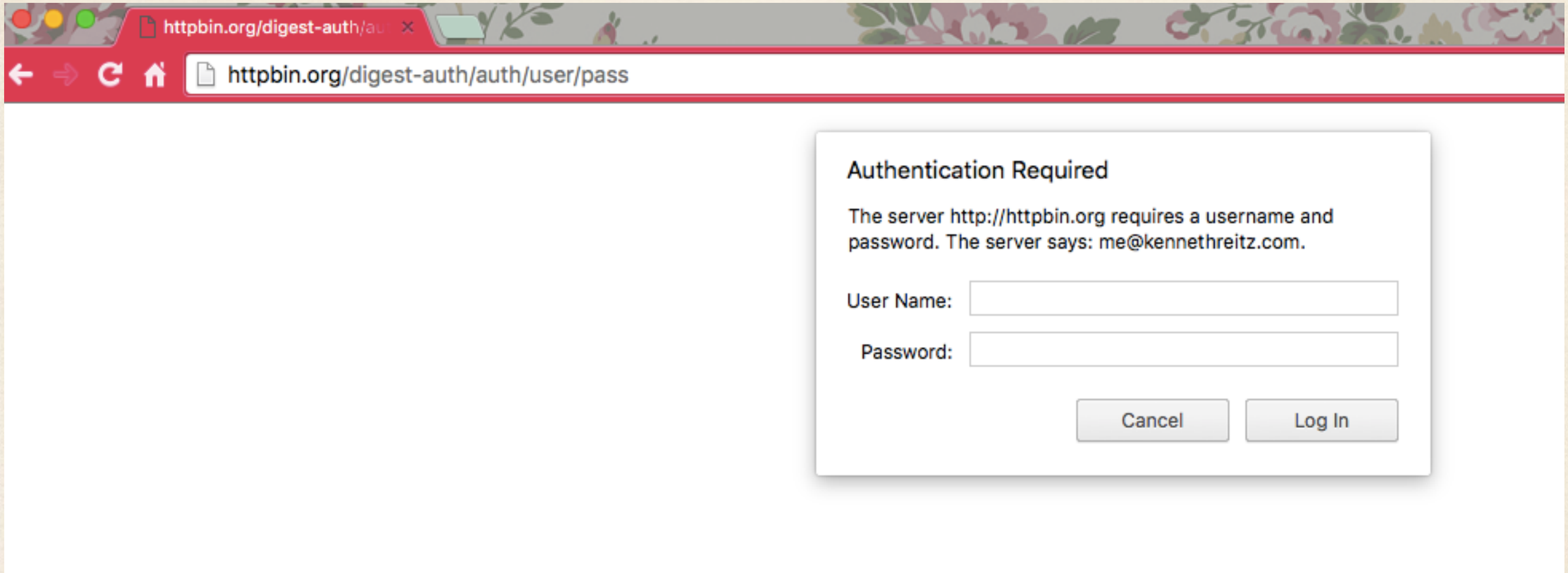
test_requests.py

Results of 1 unit test

```
[41] > /Users/susantan/Projects/requests/test_requests.py(395)test_DIGEST_HTTP_200_OK_GET()
-> r = requests.get(url, auth=auth)
(Pdb++) url
'http://127.0.0.1:63720/digest-auth/auth/user/pass'
```

What is this url http://127.0.0.01:73720/digest-auth/auth/user/pass?!

What is httpbin.org/digest-auth/auth/user/pass?



Unit test gives me the answers to both fields. I type in "user" and "pass" in both fields. The result?

Goal for today — Figure out how this code snippet works

```
>>> r = requests.get('https://api.github.com/
user', auth=('user', 'pass'))
>>> r.status_code
200
>>> r.headers['content-type']
'application/json; charset=utf8'
>>> r.encoding
'utf-8'
>>> r.text
u'{"type":"User"...'
>>> r.json()
{u'private_gists': 419, u'total_private_repos':
77}
```

Let’s look at *one* unit test

```
def test_DIGEST_HTTP_200_OK_GET(self, httpbin):

    auth = HTTPDigestAuth('user', 'pass')
    url = httpbin('digest-auth', 'auth',
'user', 'pass')

    r = requests.get(url, auth=auth)
    assert r.status_code == 200

    r = requests.get(url)
    assert r.status_code == 401

    s = requests.session()
    s.auth = HTTPDigestAuth('user', 'pass')
    r = s.get(url)
    assert r.status_code == 200
```

test_requests.py

This is the get method

```
def get(url, params=None, **kwargs):
    """Sends a GET request.

    :param url: URL for the new :class:`Request` object.
    :param params: (optional) Dictionary or bytes to be sent in the query
string for the :class:`Request`.
    :param \*\*kwargs: Optional arguments that ``request`` takes.
    :return: :class:`Response <Response>` object
    :rtype: requests.Response
    """

    kwargs.setdefault('allow_redirects', True)
    return request('get', url, params=params, **kwargs)
```

What's happening here?

- Set default dict of key-value pairs to allow redirects by default
- Returns a request. What's a request?

api.py

This is the request method

```
def request(method, url, **kwargs):
    """Constructs and sends a :class:`Request <Request>`.

    :param method: method for the new :class:`Request` object.
    :param url: URL for the new :class:`Request` object.
    :param params: (optional) Dictionary or bytes to be sent in the query string for the :class:`Request`.
    :param data: (optional) Dictionary, bytes, or file-like object to send in the body of the :class:`Request`.
    :param json: (optional) json data to send in the body of the :class:`Request`.
    :param headers: (optional) Dictionary of HTTP Headers to send with the :class:`Request`.
    :param cookies: (optional) Dict or CookieJar object to send with the :class:`Request`.
    :param files: (optional) Dictionary of ``'name': file-like-objects`` (or ``{'name': ('filename', fileobj)}``)
for multipart encoding upload.
    :param auth: (optional) Auth tuple to enable Basic/Digest/Custom HTTP Auth.
    :param timeout: (optional) How long to wait for the server to send data
        before giving up, as a float, or a :ref:`(connect timeout, read
        timeout) <timeouts>` tuple.
    :type timeout: float or tuple
    :param allow_redirects: (optional) Boolean. Set to True if POST/PUT/DELETE redirect following is allowed.
    :type allow_redirects: bool
    :param proxies: (optional) Dictionary mapping protocol to the URL of the proxy.
    :param verify: (optional) whether the SSL cert will be verified. A CA_BUNDLE path can also be provided.
Defaults to ``True``.
    :param stream: (optional) if ``False``, the response content will be immediately downloaded.
    :param cert: (optional) if String, path to ssl client cert file (.pem). If Tuple, ('cert', 'key') pair.
    :return: :class:`Response <Response>` object
    :rtype: requests.Response

    Usage::

      >>> import requests
      >>> req = requests.request('GET', 'http://httpbin.org/get')
      <Response [200]>
    """

    # By using the 'with' statement we are sure the session is closed, thus we
    # avoid leaving sockets open which can trigger a ResourceWarning in some
    # cases, and look like a memory leak in others.
    with sessions.Session() as session:
        return session.request(method=method, url=url, **kwargs)
```

api.py

This is the *same* request method without docstrings or comments

```
def request(method, url, **kwargs):

    with sessions.Session() as session:
        return session.request(method=method, url=url, **kwargs)
```

What are sessions??

```python
class Session(SessionRedirectMixin):
    """A Requests session.

    Provides cookie persistence, connection-pooling, and configuration.

    Basic Usage::

      >>> import requests
      >>> s = requests.Session()
      >>> s.get('http://httpbin.org/get')
      <Response [200]>

    Or as a context manager::

      >>> with requests.Session() as s:
      >>>     s.get('http://httpbin.org/get')
      <Response [200]>
    """

    __attrs__ = [
        'headers', 'cookies', 'auth', 'proxies', 'hooks', 'params', 'verify',
        'cert', 'prefetch', 'adapters', 'stream', 'trust_env',
        'max_redirects',
    ]

    def __init__(self):

        #: A case-insensitive dictionary of headers to be sent on each
        #: :class:`Request <Request>` sent from this
        #: :class:`Session <Session>`.
        self.headers = default_headers()

        #: Default Authentication tuple or object to attach to
        #: :class:`Request <Request>`.
        self.auth = None

        #: Dictionary mapping protocol or protocol and host to the URL of the proxy
        #: (e.g. {'http': 'foo.bar:3128', 'http://host.name': 'foo.bar:4012'}) to
        #: be used on each :class:`Request <Request>`.
        self.proxies = {}

        #: Event-handling hooks.
        self.hooks = default_hooks()

        #: Dictionary of querystring data to attach to each
        #: :class:`Request <Request>`. The dictionary values may be lists for
        #: representing multivalued query parameters.
        self.params = {}

        #: Stream response content default.
        self.stream = False

        #: SSL Verification default.
        self.verify = True

        #: SSL certificate default.
        self.cert = None

        ....
        ....
        .....
        .....
```

Really long class definition of “Sessions”

sessions.py

What are sessions??

Advanced Usage

This document covers some of Requests more advanced features.

Session Objects

The Session object allows you to persist certain parameters across requests. It also persists cookies across all requests made from the Session instance, and will use `urllib3`'s connection pooling. So if you're making several requests to the same host, the underlying TCP connection will be reused, which can result in a significant performance increase (see HTTP persistent connection).

A Session object has all the methods of the main Requests API.

Let's persist some cookies across requests:

```
s = requests.Session()

s.get('http://httpbin.org/cookies/set/sessioncookie/123456789')
r = s.get("http://httpbin.org/cookies")

print(r.text)
# '{"cookies": {"sessioncookie": "123456789"}}'
```

Sessions can also be used to provide default data to the request methods. This is done by providing data to the properties on a Session object:

```
s = requests.Session()
s.auth = ('user', 'pass')
s.headers.update({'x-test': 'true'})

# both 'x-test' and 'x-test2' are sent
s.get('http://httpbin.org/headers', headers={'x-test2': 'true'})
```

What's a session?

- an object that persists parameters across requests
- makes use of urllib3's connection pooling
- has all methods of request API
- provides default data to request object
- note: requests has well written docs

This is the *same* request method without docstrings or comments

```
def request(method, url, **kwargs):

    with sessions.Session() as session:
        return session.request(method=method, url=url, **kwargs)
```

What is this request() in Session class?

api.py

What is this request() in Session class?

```
def request(self, method, url,
        params=None,
        data=None,
        headers=None,
        cookies=None,
        files=None,
        auth=None,
        timeout=None,
        allow_redirects=True,
        proxies=None,
        hooks=None,
        stream=None,
        verify=None,
        cert=None,
        json=None):

        # Create the Request.
        req = Request(
            method = method.upper(),
            url = url,
            headers = headers,
            files = files,
            data = data or {},
            json = json,
            params = params or {},
            auth = auth,
            cookies = cookies,
            hooks = hooks,
        )
        prep = self.prepare_request(req)

        proxies = proxies or {}

        settings = self.merge_environment_settings(
            prep.url, proxies, stream, verify, cert
        )

        # Send the request.
        send_kwargs = {
            'timeout': timeout,
            'allow_redirects': allow_redirects,
        }
        send_kwargs.update(settings)
        resp = self.send(prep, **send_kwargs)

        return resp
```

sessions.py

```python
def request(self, method, url,
        params=None,
        data=None,
        headers=None,
        cookies=None,
        files=None,
        auth=None,
        timeout=None,
        allow_redirects=True,
        proxies=None,
        hooks=None,
        stream=None,
        verify=None,
        cert=None,
        json=None):

    # Create the Request.
    req = Request(
        method = method.upper(),
        url = url,
        headers = headers,
        files = files,
        data = data or {},
        json = json,
        params = params or {},
        auth = auth,
        cookies = cookies,
        hooks = hooks,
    )
    prep = self.prepare_request(req)

    proxies = proxies or {}

    settings = self.merge_environment_settings(
        prep.url, proxies, stream, verify, cert
    )

    # Send the request.
    send_kwargs = {
        'timeout': timeout,
        'allow_redirects': allow_redirects,
    }
    send_kwargs.update(settings)
    resp = self.send(prep, **send_kwargs)

    return resp
```

1

2

3

4

What's happening in this code?

1. create request
2. create prepare request object "prep"
3. send request
4. return response

```python
# Create the Request.
req = Request(
    method = method.upper(),
    url = url,
    headers = headers,
    files = files,
    data = data or {},
    json = json,
    params = params or {},
    auth = auth,
    cookies = cookies,
    hooks = hooks,
)
```

sessions.py

Let’s dissect requests/sessions.py

1.create request

This is Request class definition

```
class Request(RequestHooksMixin):
    """A user-created :class:`Request <Request>` object.

    Used to prepare a :class:`PreparedRequest <PreparedRequest>`, which is sent to the server.

    :param method: HTTP method to use.
    :param url: URL to send.
    :param headers: dictionary of headers to send.
    :param files: dictionary of {filename: fileobject} files to multipart upload.
    :param data: the body to attach to the request. If a dictionary is provided, form-encoding will take place.
    :param json: json for the body to attach to the request (if files or data is not specified).
    :param params: dictionary of URL parameters to append to the URL.
    :param auth: Auth handler or (user, pass) tuple.
    :param cookies: dictionary or CookieJar of cookies to attach to this request.
    :param hooks: dictionary of callback hooks, for internal usage.

    Usage::

      >>> import requests
      >>> req = requests.Request('GET', 'http://httpbin.org/get')
      >>> req.prepare()
      <PreparedRequest [GET]>

    """
    def __init__(self, method=None, url=None, headers=None, files=None,
        data=None, params=None, auth=None, cookies=None, hooks=None, json=None):

        # Default empty dicts for dict params.
        data = [] if data is None else data
        files = [] if files is None else files
        headers = {} if headers is None else headers
        params = {} if params is None else params
        hooks = {} if hooks is None else hooks

        self.hooks = default_hooks()
        for (k, v) in list(hooks.items()):
            self.register_hook(event=k, hook=v)

        self.method = method
        self.url = url
        self.headers = headers
        self.files = files
        self.data = data
        self.json = json
        self.params = params
        self.auth = auth
        self.cookies = cookies

    def __repr__(self):
        return '<Request [%s]>' % (self.method)

    def prepare(self):
        """Constructs a :class:`PreparedRequest <PreparedRequest>` for transmission and returns it."""
        p = PreparedRequest()
        p.prepare(
            method=self.method,
            url=self.url,
            headers=self.headers,
            files=self.files,
            data=self.data,
            json=self.json,
            params=self.params,
            auth=self.auth,
            cookies=self.cookies,
            hooks=self.hooks,
        )
        return p
```

request arguments to create request() object

models.py

```python
def request(self, method, url,
        params=None,
        data=None,
        headers=None,
        cookies=None,
        files=None,
        auth=None,
        timeout=None,
        allow_redirects=True,
        proxies=None,
        hooks=None,
        stream=None,
        verify=None,
        cert=None,
        json=None):

    # Create the Request.
    req = Request(
        method = method.upper(),
        url = url,
        headers = headers,
        files = files,
        data = data or {},
        json = json,
        params = params or {},
        auth = auth,
        cookies = cookies,
        hooks = hooks,
    )
    prep = self.prepare_request(req)

    proxies = proxies or {}

    settings = self.merge_environment_settings(
        prep.url, proxies, stream, verify, cert
    )

    # Send the request.
    send_kwargs = {
        'timeout': timeout,
        'allow_redirects': allow_redirects,
    }
    send_kwargs.update(settings)
    resp = self.send(prep, **send_kwargs)

    return resp
```

1

2

3

4

What's happening in this code?

1. create request ✓
2. create prepare request object "prep"
3. send request
4. return response

```python
prep = self.prepare_request(req)
```

sessions.py

Let’s dissect requests/sessions.py

2. create prepare request object “prep”

```
def prepare_request(self, request):
    ……
    p = PreparedRequest()
    p.prepare(
        method=request.method.upper(),
        url=request.url,
        files=request.files,
        data=request.data,
        json=request.json,
       headers=
        merge_setting(…),
        auth=merge_setting(auth, self.auth),
        cookies=merged_cookies,
        hooks=merge_hooks(request.hooks,
self.hooks),
        )
        return p
```

What is “PreparedRequests class”?
What is “prepare()”?

sessions.py

What are Prepared Requests?

```python
class PreparedRequest(RequestEncodingMixin, RequestHooksMixin):
    """The fully mutable :class:`PreparedRequest <PreparedRequest>` object,
    containing the exact bytes that will be sent to the server.

    Generated from either a :class:`Request <Request>` object or manually.

    Usage::

      >>> import requests
      >>> req = requests.Request('GET', 'http://httpbin.org/get')
      >>> r = req.prepare()
      <PreparedRequest [GET]>

      >>> s = requests.Session()
      >>> s.send(r)
      <Response [200]>

    """

    def __init__(self):
        #: HTTP verb to send to the server.
        self.method = None
        #: HTTP URL to send the request to.
        self.url = None
        #: dictionary of HTTP headers.
        self.headers = None
        # The `CookieJar` used to create the Cookie header will be stored here
        # after prepare_cookies is called
        self._cookies = None
        #: request body to send to the server.
        self.body = None
        #: dictionary of callback hooks, for internal usage.
        self.hooks = default_hooks()

    def prepare(self, method=None, url=None, headers=None, files=None,
        data=None, params=None, auth=None, cookies=None, hooks=None, json=None):
        """Prepares the entire request with the given parameters."""

        self.prepare_method(method)
        self.prepare_url(url, params)
        self.prepare_headers(headers)
        self.prepare_cookies(cookies)
        self.prepare_body(data, files, json)
        self.prepare_auth(auth, url)

        # Note that prepare_auth must be last to enable authentication schemes
        # such as OAuth to work on a fully prepared request.

        # This MUST go after prepare_auth. Authenticators could add a hook
        self.prepare_hooks(hooks)
```

Lots more layers of abstraction!

models.py

```python
def request(self, method, url,
        params=None,
        data=None,
        headers=None,
        cookies=None,
        files=None,
        auth=None,
        timeout=None,
        allow_redirects=True,
        proxies=None,
        hooks=None,
        stream=None,
        verify=None,
        cert=None,
        json=None):

    # Create the Request.
    req = Request(
        method = method.upper(),
        url = url,
        headers = headers,
        files = files,
        data = data or {},
        json = json,
        params = params or {},
        auth = auth,
        cookies = cookies,
        hooks = hooks,
    )
    prep = self.prepare_request(req)

    proxies = proxies or {}

    settings = self.merge_environment_settings(
        prep.url, proxies, stream, verify, cert
    )

    # Send the request.
    send_kwargs = {
        'timeout': timeout,
        'allow_redirects': allow_redirects,
    }
    send_kwargs.update(settings)
    resp = self.send(prep, **send_kwargs)

    return resp
```

1

2

3

4

What's happening in this code?

1. create request ✓
2. create prepare request object "prep" ✓
3. send request
4. return response

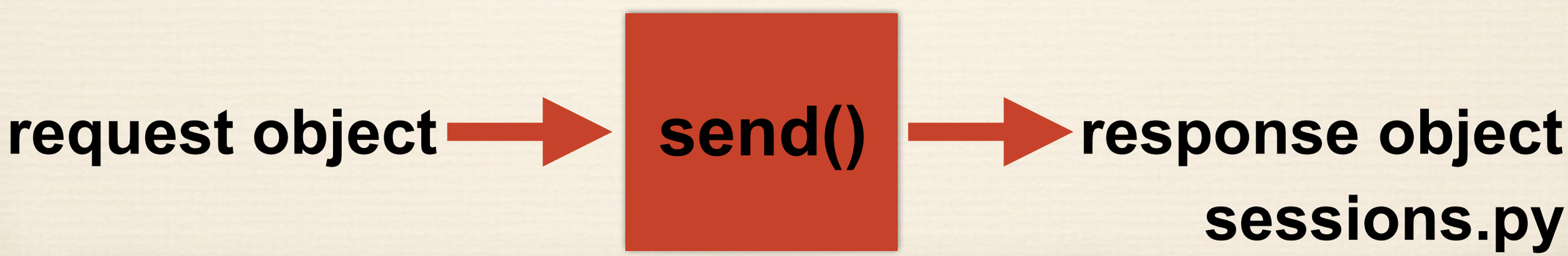
```python
resp =
self.send(prep,
**send_kwargs)

return resp
```

sessions.py

Let’s dissect requests/sessions.py

3. send request
4. return response

```
def send(self, request, **kwargs):
    """Send a given PreparedRequest."""

    ….[LONG METHOD DEFINITION HERE]….

    return r
```



sessions.py

```python
def send(self, request, **kwargs):
        """Send a given PreparedRequest."""
        # Set defaults that the hooks can utilize to ensure they
always have
        # the correct parameters to reproduce the previous request.
        kwargs.setdefault('stream', self.stream)
        kwargs.setdefault('verify', self.verify)
        kwargs.setdefault('cert', self.cert)
        kwargs.setdefault('proxies', self.proxies)

        # It's possible that users might accidentally send a Request
object.
        # Guard against that specific failure case.
        if not isinstance(request, PreparedRequest):
            raise ValueError('You can only send PreparedRequests.')

        checked_urls = set()
        while request.url in self.redirect_cache:
            checked_urls.add(request.url)
            new_url = self.redirect_cache.get(request.url)
            if new_url in checked_urls:
                break
            request.url = new_url

        # Set up variables needed for resolve_redirects and
dispatching of hooks
        allow_redirects = kwargs.pop('allow_redirects', True)
        stream = kwargs.get('stream')
        hooks = request.hooks

        # Get the appropriate adapter to use
        adapter = self.get_adapter(url=request.url)

        # Start time (approximately) of the request
        start = datetime.utcnow()

        # Send the request
        r = adapter.send(request, **kwargs)

        # [illegible] mately)
        r.elapsed = datetime.utcnow() - start

        # Response manipulation hooks
        r = dispatch_hook('response', hooks, r, **kwargs)

        # Persist cookies
        if r.history:

            # If the hooks create history then we want those cookies
too
            for resp in r.history:
                extract_cookies_to_jar(self.cookies, resp.request,
resp.raw)

        extract_cookies_to_jar(self.cookies, request, r.raw)

        # Redirect resolving generator.
        gen = self.resolve_redirects(r, request, **kwargs)

        # Resolve redirects if allowed.
        history = [resp for resp in gen] if allow_redirects else []

        # Shuffle things around if there's history.
        if history:
            # Insert the first (original) request at the start
            history.insert(0, r)
            # Get the last request made
            r = history.pop()
            r.history = history

        if not stream:
            r.content

        return r
```

3. send request
4. return response

What's an adapter?

```python
# Get the appropriate adapter to use
adapter = self.get_adapter(url=request.url)

# Start time (approximately) of the request
start = datetime.utcnow()

# Send the request
r = adapter.send(request, **kwargs)
```

What is the send method doing in adapters.py?

sessions.py

What's an adapter?

"This adapter provides the default Requests interaction with HTTP and HTTPS using the powerful urllib3 library."

—"Transport Adapters" in requests advanced docs

This is HTTPAdapter class, the interface for urllib3

```
class HTTPAdapter(BaseAdapter):
    """
    The built-in HTTP Adapter for urllib3.

    Provides a general-case interface for Requests sessions to contact HTTP and
    HTTPS urls by implementing the Transport Adapter interface. This class will
    usually be created by the :class:`Session <Session>` class under the
    covers.

    :param pool_connections: The number of urllib3 connection pools to cache.
    :param pool_maxsize: The maximum number of connections to save in the pool.
    :param int max_retries: The maximum number of retries each connection
        should attempt. Note, this applies only to failed DNS lookups, socket
        connections and connection timeouts, never to requests where data has
        made it to the server. By default, Requests does not retry failed
        connections. If you need granular control over the conditions under
        which we retry a request, import urllib3's ``Retry`` class and pass
        that instead.
    :param pool_block: Whether the connection pool should block for connections.

    Usage::

      >>> import requests
      >>> s = requests.Session()
      >>> a = requests.adapters.HTTPAdapter(max_retries=3)
      >>> s.mount('http://', a)
    """
```

adapters.py

the imports at top of requests/adapters.py

```
"""
requests.adapters
~~~~~~~~~~~~~~~~~

This module contains the transport adapters that Requests uses to define
and maintain connections.
"""
import os.path
import socket

from .models import Response
from .packages.urllib3.poolmanager import PoolManager, proxy_from_url
from .packages.urllib3.response import HTTPResponse
from .packages.urllib3.util import Timeout as TimeoutSauce
from .packages.urllib3.util.retry import Retry
from .compat import urlparse, basestring
from .utils import (DEFAULT_CA_BUNDLE_PATH, get_encoding_from_headers,
                    prepend_scheme_if_needed, get_auth_from_url, urldefragauth,
                    select_proxy)
from .structures import CaseInsensitiveDict
from .packages.urllib3.exceptions import ClosedPoolError
from .packages.urllib3.exceptions import ConnectTimeoutError
from .packages.urllib3.exceptions import HTTPError as _HTTPError
from .packages.urllib3.exceptions import MaxRetryError
from .packages.urllib3.exceptions import NewConnectionError
from .packages.urllib3.exceptions import ProxyError as _ProxyError
from .packages.urllib3.exceptions import ProtocolError
from .packages.urllib3.exceptions import ReadTimeoutError
from .packages.urllib3.exceptions import SSLError as _SSLError
from .packages.urllib3.exceptions import ResponseError
from .cookies import extract_cookies_to_jar
from .exceptions import (ConnectionError, ConnectTimeout, ReadTimeout, SSLError,
                         ProxyError, RetryError)
from .auth import _basic_auth_str

DEFAULT_POOLBLOCK = False
DEFAULT_POOLSIZE = 10
DEFAULT_RETRIES = 0
DEFAULT_POOL_TIMEOUT = None
```

urllib3 is imported here

adapters.py

```
def send(self, request, stream=False, timeout=None, verify=True, cert=None, proxies=None):
    """Sends PreparedRequest object. Returns Response object.

    :param request: The :class:`PreparedRequest <PreparedRequest>` being sent.
    :param stream: (optional) Whether to stream the request content.
    :param timeout: (optional) How long to wait for the server to send
        data before giving up, as a float, or a :ref:`(connect timeout,
        read timeout) <timeouts>` tuple.
    :type timeout: float or tuple
    :param verify: (optional) Whether to verify SSL certificates.
    :param cert: (optional) Any user-provided SSL certificate to be trusted.
    :param proxies: (optional) The proxies dictionary to apply to the request.
    """

    conn = self.get_connection(request.url, proxies)

    self.cert_verify(conn, request.url, verify, cert)
    url = self.request_url(request, proxies)
    self.add_headers(request)

    chunked = not (request.body is None or 'Content-Length' in request.headers)

    if isinstance(timeout, tuple):
        try:
            connect, read = timeout
            timeout = TimeoutSauce(connect=connect, read=read)
        except ValueError as e:
            # this may raise a string formatting error.
            err = ("Invalid timeout {0}. Pass a (connect, read) "
                   "timeout tuple, or a single float to set "
                   "both timeouts to the same value".format(timeout))
            raise ValueError(err)
    else:
        timeout = TimeoutSauce(connect=timeout, read=timeout)

    try:
        if not chunked:
            resp = conn.urlopen(
                method=request.method,
                url=url,
                body=request.body,
                headers=request.headers,
                redirect=False,
                assert_same_host=False,
                preload_content=False,
                decode_content=False,
                retries=self.max_retries,
                timeout=timeout
            )

        # Send the request.
        else:
            if hasattr(conn, 'proxy_pool'):
                conn = conn.proxy_pool

            low_conn = conn._new_conn(timeout=DEFAULT_POOL_TIMEOUT)

            try:
                low_conn.putrequest(request.method,
                                    url,
                                    skip_accept_encoding=True)

                for header, value in request.headers.items():
                    low_conn.putheader(header, value)

                low_conn.endheaders()

                for i in request.body:
                    low_conn.send(hex(len(i))[2:].encode('utf-8'))
                    low_conn.send(b'\r\n')
                    low_conn.send(i)
                    low_conn.send(b'\r\n')
                low_conn.send(b'0\r\n\r\n')

                # Receive the response from the server
                try:
                    # For Python 2.7+ versions, use buffering of HTTP
                    # responses
                    r = low_conn.getresponse(buffering=True)
                    except TypeError:
                    # For compatibility with Python 2.6 versions and back
                    r = low_conn.getresponse()

                resp = HTTPResponse.from_httplib(
                    r,
                    pool=conn,
                    connection=low_conn,
                    preload_content=False,
                    decode_content=False
                )
            except:
                # If we hit any problems here, clean up the connection.
                # Then, reraise so that we can handle the actual exception.
                low_conn.close()
                raise

    except (ProtocolError, socket.error) as err:
        raise ConnectionError(err, request=request)

    except MaxRetryError as e:
        if isinstance(e.reason, ConnectTimeoutError):
            # TODO: Remove this in 3.0.0: see #2811
            if not isinstance(e.reason, NewConnectionError):
                raise ConnectTimeout(e, request=request)

        if isinstance(e.reason, ResponseError):
            raise RetryError(e, request=request)

        raise ConnectionError(e, request=request)

    except ClosedPoolError as e:
        raise ConnectionError(e, request=request)

    except _ProxyError as e:
        raise ProxyError(e)

    except (_SSLError, _HTTPError) as e:
        if isinstance(e, _SSLError):
            raise SSLError(e, request=request)
        elif isinstance(e, ReadTimeoutError):
            raise ReadTimeout(e, request=request)
        else:
            raise
```

Size 5 font. This is the definition of send() method. Over 100 lines long and it can't fit this slide. Exercise left to reader to read thru this send() method.

adapters.py

Let’s place a debugger in adapters.py and run the same unit test again.

```
def send(self, request, stream=False, timeout=None, verify=True, cert=None,
proxies=None):
    …
    import pdp
    pdb.set_trace()
    return self.build_response(request, resp)
```

adapters.py

Run this same unit test on command line

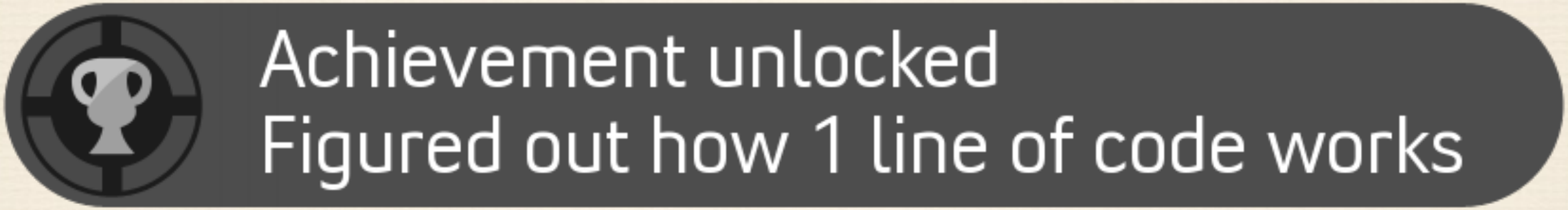
py.test test_requests.py::TestRequests::test_DIGEST_HTTP_200_OK_GET

Use pytest debugger to see output of send() method

```
[48] > /Users/susantan/Projects/requests/requests/
adapters.py(455)send()
-> return self.build_response(request, resp)

(Pdb++) request.url
'http://127.0.0.1:58948/digest-auth/auth/user/pass'

(Pdb++) resp
<requests.packages.urllib3.response.HTTPResponse object
at 0x102c15110>

(Pdb++) our_version_of_response_object =
self.build_response(request, resp)

(Pdb++) our_version_of_response_object.json()
{u'authenticated': True, u'user': u'user'}

(Pdb++) our_version_of_response_object.status_code
200
```

Goal for today — Figure out how this code snippet works

```
>>> r = requests.get('https://api.github.com/
user', auth=('user', 'pass'))
>>> r.status_code
200
>>> r.headers['content-type']
'application/json; charset=utf8'
>>> r.encoding
'utf-8'
>>> r.text
u'{"type":"User"...'
>>> r.json()
{u'private_gists': 419, u'total_private_repos':
77}
```


Achievement unlocked
Figured out how 1 line of code works

In summary, how does “request.get()” work?

```
def test_DIGEST_HTTP_200_OK_GET(self, httpbin):

    auth = HTTPDigestAuth('user', 'pass')
    url = httpbin('digest-auth', 'auth',
'user', 'pass')

    r = requests.get(url, auth=auth)
    assert r.status_code == 200

    r = requests.get(url)
    assert r.status_code == 401

    s = requests.session()
    s.auth = HTTPDigestAuth('user', 'pass')
    r = s.get(url)
    assert r.status_code == 200
```

test_requests.py

This is the get method

```
def get(url, params=None, **kwargs):

    kwargs.setdefault('allow_redirects', True)
    return request('get', url, params=params,
**kwargs)
```

This is the request method

```
def request(method, url, **kwargs):

    with sessions.Session() as session:
        return session.request(method=method, url=url,
**kwargs)
```

api.py

```python
def request(self, method, url,
        params=None,
        data=None,
        headers=None,
        cookies=None,
        files=None,
        auth=None,
        timeout=None,
        allow_redirects=True,
        proxies=None,
        hooks=None,
        stream=None,
        verify=None,
        cert=None,
        json=None):

    # Create the Request.
    req = Request(
        method = method.upper(),
        url = url,
        headers = headers,
        files = files,
        data = data or {},
        json = json,
        params = params or {},
        auth = auth,
        cookies = cookies,
        hooks = hooks,
    )
    prep = self.prepare_request(req)

    proxies = proxies or {}

    settings = self.merge_environment_settings(
        prep.url, proxies, stream, verify, cert
    )

    # Send the request.
    send_kwargs = {
        'timeout': timeout,
        'allow_redirects': allow_redirects,
    }
    send_kwargs.update(settings)
    resp = self.send(prep, **send_kwargs)

    return resp
```

1

2

3

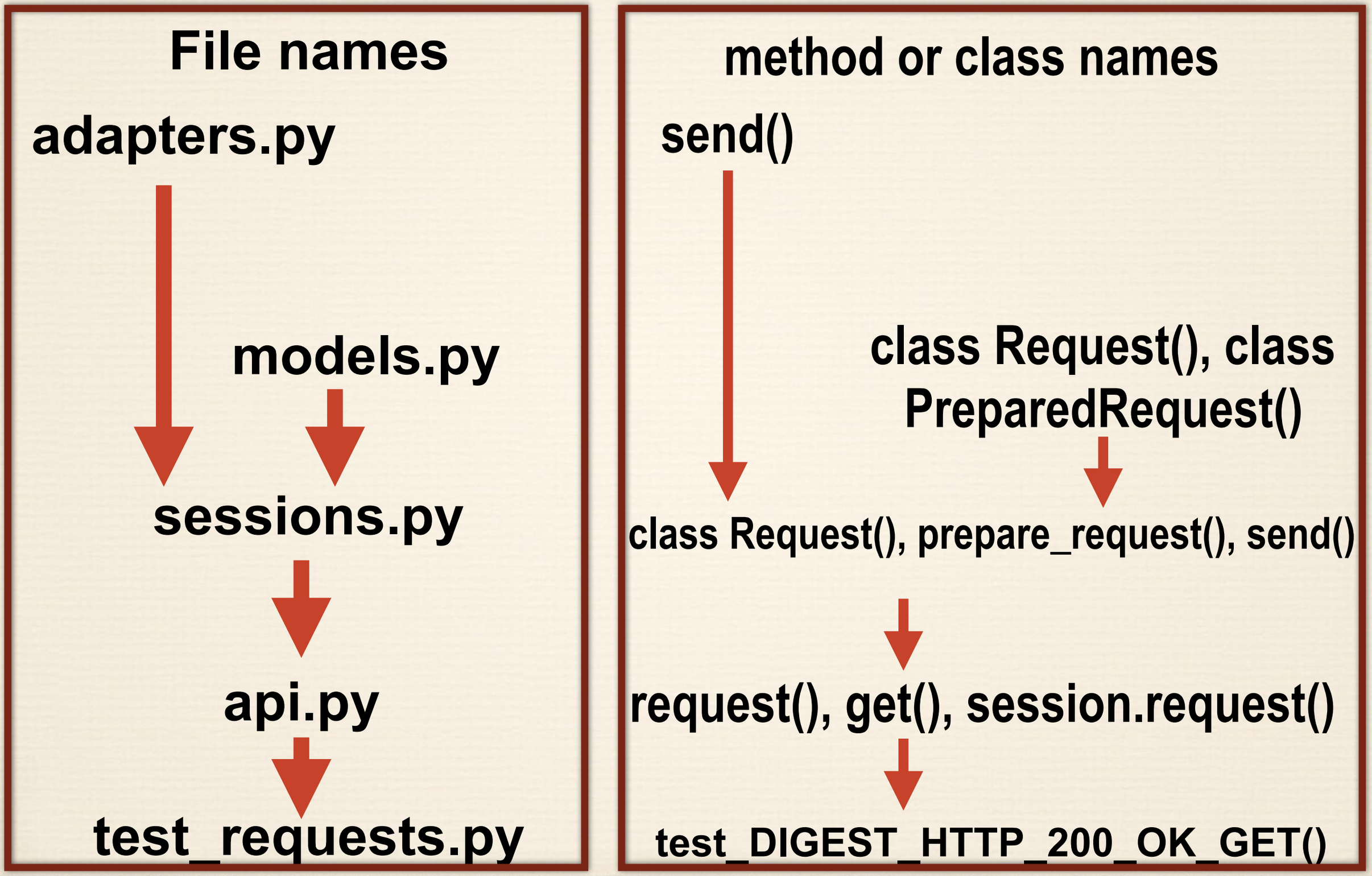
4

What's happening in this code?

1. create request ✓
2. create prepare request object "prep" ✓
3. send request ✓
4. return response ✓

sessions.py

A mental map of files and associated function calls



No walkthrough of a codebase is the same for any person. An Alternative —

```
In [4]: import requests
In [4]: resp = requests.post('http://httpbin.org/post',
data={'name':'Susan'})
[14] > /Users/susantan/Projects/requests/requests/
adapters.py(346)send()
```

```
342
343             import pdb
344             pdb.set_trace()
345
346  ->         conn = self.get_connection(request.url, proxies)
347
348             self.cert_verify(conn, request.url, verify, cert)
349             url = self.request_url(request, proxies)
350             self.add_headers(request)
```

Set breakpoints in adapters.py

Takeaways

```
r = requests.get('https://api.github.com/
user', auth=('user', 'pass'))
```

We ***really*** know "*requests.get(url)*" works in great depth.

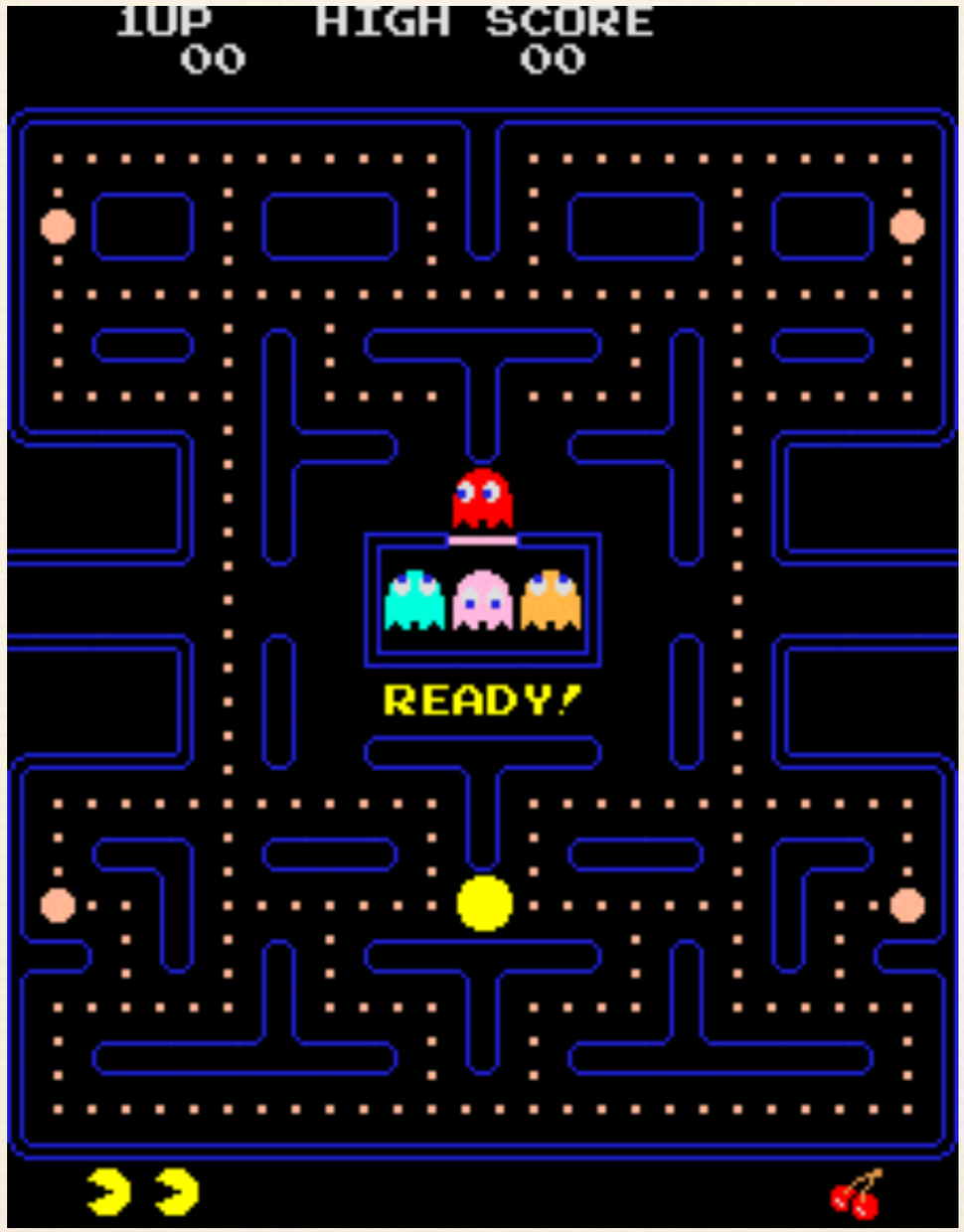
What to do when you get really stuck on figuring out codebase?

- Talk to core devs or maintainers
- git blame

What to do when you get really stuck on figuring out codebase?

Use your favorite python shell and debugger to explore a small code sample.

Call to Action:
more codebase walkthroughs



Hope this helps.

Where to reach me:

@ArcTanSusan on Twitter

San Francisco, CA