Python requests

Getting Started with

Python requests

Welcome!

World's fastest introduction

- Start with theory and simple demos
- Advance into real-life usage examples
- Explore professional-grade business tools
- I recommend NOT following along
  - Ask me questions now, watch recording later
  - Please stay on topic and consider other attendees

Python requests

1.1 HTTP Introduction

HTTP for Kindergarteners

- Stuff Transport Protocol (Russ White)
  - Data exchange on the web
- Variety of operations/methods
  - Create, read, update, and delete (CRUD)
- Variety of payload types and real-life applications
  - Browsing, API exchanges, file transfer, etc.

HTTP Components

- A client sends requests, a server sends responses
- Both messages have:
  - A request line (request) or status line (response)
  - Headers
  - An optional body, separated by blank line

Common HTTP Methods/Operations

- GET: Read a resource
- POST: Create a resource
- PUT: Update a resource
- DELETE: Delete a resource (together, CRUD)

- Less common
  - HEAD: Get the headers only; no body
  - PATCH: Update part of a resource

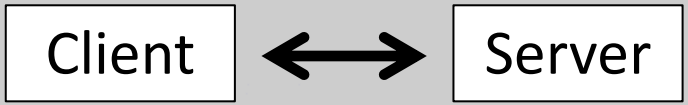
HTTP status codes – "successes"

- 100 – 199: Continue
- 200 – 299: Success
  - 200: OK
  - 201: Created (new resource)
  - 202: Accepted (asynchronous OK)
- 300 – 399: Redirect
  - 301: Moved Permanently
  - 307: Temporary Redirect

HTTP status codes - errors

- 400 – 499: Client errors
  - 400: Bad Request
  - 401: Unauthorized (authentication)
  - 403: Forbidden (authorization)
  - 404: Not Found
  - 405: Method not allowed
- 500 – 599: Server errors
  - 500: Internal Server Error
  - 501: Not implemented

HTTP transaction example - success



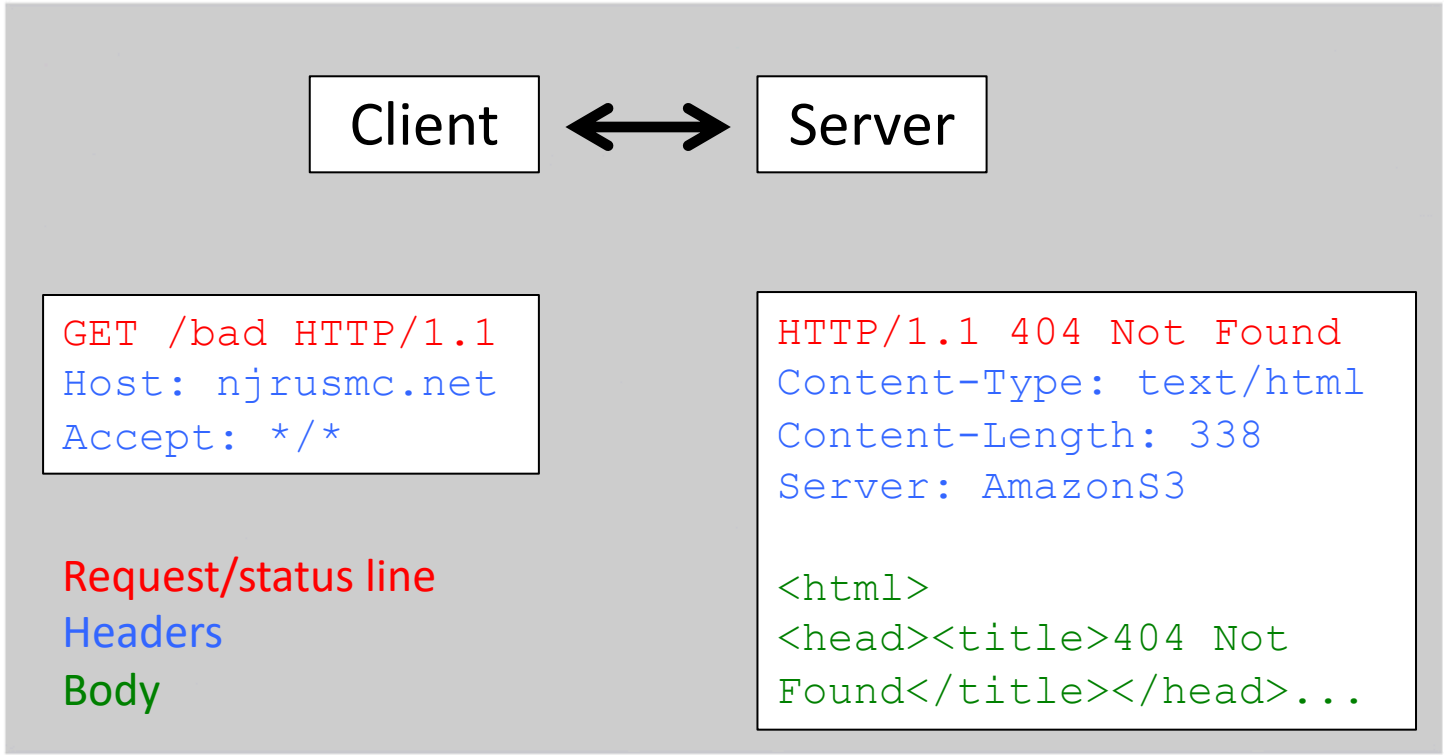

```
GET / HTTP/1.1
Host: njrusmc.net
Accept: */*
```

Request/status line
Headers
Body

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 1796
Server: AmazonS3

<!DOCTYPE html>
<html
xmlns="http://www.w3.org
/1999/xhtml">
```

HTTP transaction example - error



Python requests

1.2 Introduction to requests

https://requests.readthedocs.io

What is Python requests?

- Python requests is an HTTP client
  - Easy way to interact with web services
  - Universally applicable across use-cases

```
>>> import requests
>>> resp = requests.get("http://njrusmc.net")
>>> resp.status_code
200
>>> len(resp.headers)
8
>>> resp.text
'<!DOCTYPE html>\n<html xmlns=...'
```

Issuing HTTP requests

- Two main options
- Generic function:
  - requests.request(method="get")
  - requests.request(method="post")
- Specific functions:
  - requests.get(), requests.post(), requests.put(), etc.
  - Simple wrappers for the generic function

Specifying headers

- Headers are key-value pairs
  - Great fit for a Python dictionary
  - Often carries API keys/tokens, accepted encodings, etc.
  - Very application-dependent

```
>>> import requests
>>> my_headers = {"Accept": "text/html"}
>>> resp = requests.get("http://njrusmc.net",
headers=my_headers)
>>> resp.headers["Content-Type"]
'text/html'
```

Specifying query parameters

- Query parameters are key-value pairs
  - Great fit for a Python dictionary (again)
  - Appended to URL in "?key=value&key=value" format
  - Often not required; just a selection/filtering technique

```
>>> import requests
>>> params = {"limit": 5, "type": "tech"}
>>> resp = requests.get("http://njrusmc.net",
params=params)
>>> resp.request.url
'http://njrusmc.net/?limit=5&type=tech'
```

Specifying the body

- Can use "data" to specify key-value pairs or plain-text
- Other techniques exist (discussed later)

```
>>> import requests
>>> body = {"name": "nick", "kids": 2}
>>> resp = requests.get("http://njrusmc.net",
data=body)
>>> resp.request.body
'name=nick&kids=2'

>>> resp = requests.get("http://njrusmc.net",
data="hi!")
>>> resp.request.body
'hi!'
```

It's not over!

- We're just getting started
- Many more options to be revealed in demos
  - SSL verification
  - Long-lived TCP sessions
  - Content caching
  - REST API interaction
  - File upload/download

Python requests

1.3 Installation and operations

Installing via pip

- Python package manager
- Works like yum/apt for Linux distributions
- pip install requests (includes dependencies)

```
$ pip install requests
Collecting requests
  Downloading requests-2.23.0-py2.py3-none-any.whl
(58 kB)
(...)
Installing collected packages: requests
Successfully installed requests-2.23.0
```

Using requests in a script

- Typically, just "import requests" and go

```
$ cat simple_script.py
import requests
resp = requests.get("http://njrusmc.net")
print(resp.status_code, resp.reason)
print(resp.text)

$ python simple_script.py
200 OK
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
(...)
```

Python logging

- Set basic configuration
  - Timestamp formatting
  - Log level
- Create a "logger" object
- Call the logger's methods to write logs
  - logger.info(msg), logger.debug(msg), etc.

Python debugger (pdb)

- Four core actions
  - **list (l):** reveal the code +/- 5 lines
  - **next (n):** step over function
  - **step (s):** step into function
  - **continue (c):** run until next breakpoint

- Add breakpoint
  - "breakpoint()" in py3.7+
  - "import pdb; pdb.set_trace()" in py3.6-

Python requests

2.1 HTTP Content Caching

Purpose of Caching

- Client benefits
  - Download once, consume many times
  - Fast access to existing content
  - Happy customers!
- Server benefits
  - Less load on web servers
  - Less transit bandwidth consumed
  - Lower capex/opex for carriers

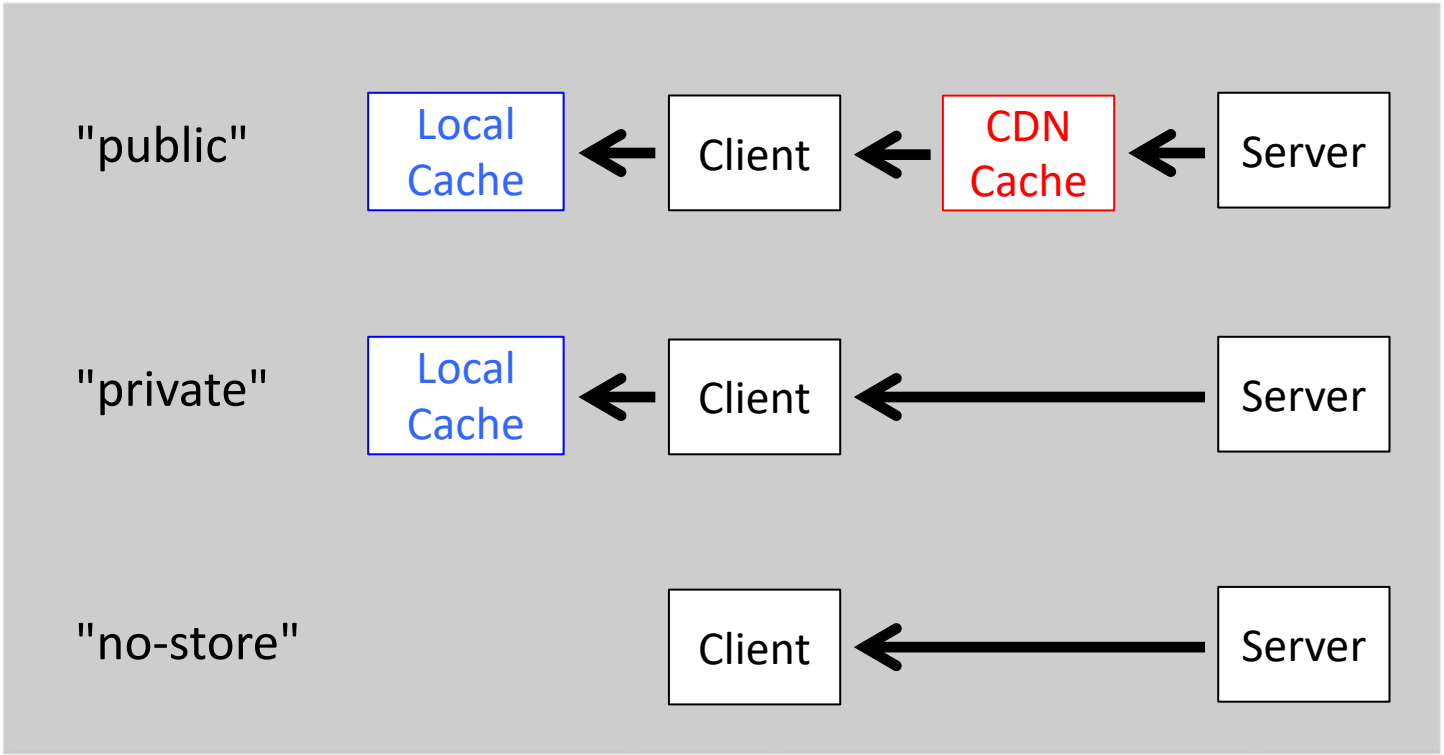
Types of Caches

- Two general types
- Local cache on the client
  - Temporary browser files
  - Application-specific (e.g. CacheControl package)
- Content Delivery Network (CDN)
  - Centralized, strategically placed public caches
  - Netflix, CloudFlare, or a corporate DMZ/data center

How to Enable?

- HTTP response header of "Cache-Control"
  - Many minor options
  - **Not a deep dive!**
  - Test files: http://njrusmc.net/cache/cache.html
- "public": public or private caches
- "private": private (local) caches only
- "no-store": no caches
- Omitted: heuristic freshness via on last-modified time

Caching Options Visualized



Python requests

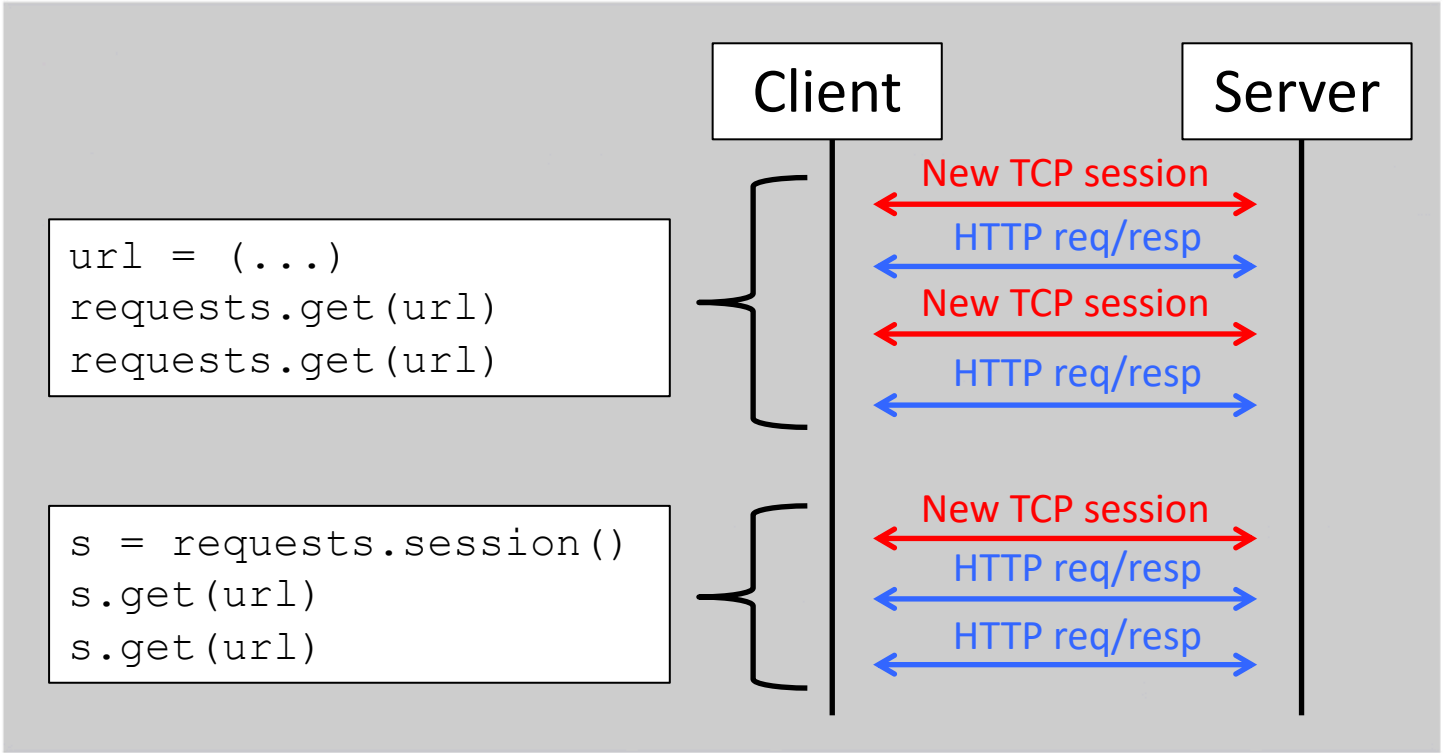
2.2 Using CacheControl

https://cachecontrol.readthedocs.io

Python CacheControl package

- pip install CacheControl
- Uses the concept of a "session"
  - HTTP is stateless, but TCP isn't
  - Can create a long-lived TCP session
  - Improves performance
- Using a "session" is required for CacheControl
  - ... and handy for cookie-based REST APIs

The requests "session"



What about caching?

- Import CacheControl class from cachecontrol module
- Create regular requests session
- Pass into CacheControl
- Use standard requests-style methods

```
import requests
from cachecontrol import CacheControl

sess = requests.session()
cached_sess = CacheControl(sess)
resp1 = cached_sess.get(url) # not cached
resp2 = cached_sess.get(url) # is cached*
```

Python requests

2.3 REST Refresher

What is REST?

- Representational State Transfer
- An architectural style for web service interaction
- Not ...
  - A protocol
  - A framework
  - A language
  - A package

REST Constraints

- To be RESTful, a system must have/be:
  - Uniform interface: resources directly accessible
  - Client-server: separate user intf from data storage
  - Stateless: requests are independent; no server state
  - Cacheable: we just talked about this ☺
  - Layered system: hierarchical servers, load balancers, etc.
  - Code on demand (optional): server extends functionality of client at runtime by sending code (JavaScript, etc.)

Some conclusions

- Most constraints are met using HTTP
  - Don't HAVE to use HTTP, but it's a natural fit
  - One URI per resource → HTTP URL in a request
  - CRUD operations easily map to HTTP methods
  - Note: return codes can be tricky
    - Can use HTTP codes (easy and common)
    - Can use HTTP 200 with embedded status code
      - Separates transport status from app/resource status

Python requests

2.4 A caveat; "data" vs. "json"

HTTP body: data vs. json

- Relates to HTTP bodies
- If "data" is a dict, unpacks to HTML key/value pairs
- Automatically adds proper Content-Type header
- Commonly used for webform interaction
- Can also be plain-text (no transformation/header)

Using "data" to include HTTP body

```
>>> import requests
>>> resp = requests.post(
    "http://njrusmc.net",
    data={"name": "nick", "age": 34}
)

# request failed, but it doesn't matter
>>> resp.request.body
'name=nick&age=34'
>>> resp.request.headers
{'Content-Type': 'application/x-www-form-
urlencoded'}
```

What about JSON?

- Looks similar at a glance
- Must be a dictionary, but serializes into JSON as text
- Automatically adds proper Content-Type header
- Commonly used for interacting with REST APIs

Using "json" to include HTTP body

```
>>> import requests
>>> resp = requests.post(
    "http://njrusmc.net",
    json={"name": "nick", "age": 34}
)

# Very different headers and body!
>>> resp.request.body
b'{"name": "nick", "age": 34}'
>>> resp.request.headers
{'Content-Type': 'application/json'}
```

Python requests

2.5 Cisco SD-WAN demo

https://developer.cisco.com/sdwan/learn/

Python requests

3.1 Some fun examples; pokeapi

https://pokeapi.co/docs/v2

What is pokeapi?

- A free, read-only (GET) REST API
- Features 1,000+ Pokémon characters
- Tons of data, JSON-encoded, and well-documented
- Hosted at https://pokeapi.co/api/v2

Python requests

3.2 Business case: tweeter-lite

https://github.com/nickrusso42518/tweeter-lite

What is tweeter-lite?

- Validates a Twitter message (called tweets)
- Start with library of known-good tweets
- Validates two important components
  - Message is not too long
  - URL in message is reachable
- Full version is more complex (but irrelevant today)

Handling redirects

- Automatically follows HTTP redirects (code 3XX)
- Some notes:
  - Set explicitly with allow_redirects=True/False
  - Is disabled for HTTP HEAD requests
  - Tracks redirected requests in "history" list
- Disabling might improve security (validate each link)

Python requests

4.1 Business case: py-auphonic

https://github.com/nickrusso42518/py-auphonic

What is py-auphonic?

- Python client for www.auphonic.com REST API
- Used for post-production of audio files
- Great for VoD/AoD, podcasts, and music
- Well-documented, reliable REST API
  - https://auphonic.com/help/api/

My workflow - Preset

- Preset is a template for producing audio files
- Many options
  - Encoding type (mp3, wav, etc.)
  - Noise cancellation
  - Loudness adjustment
  - Audio smoothing
  - Metadata (title, artist, album, track)

My workflow - Production

- A production is a container for files
- Can do one file per production, or many ("batch")
- Basic steps
  - **Create** production from preset
  - **Upload** file(s) into production
  - **Produce** the audio
  - **Download** the resulting file

Python requests

4.2 Using "py-auphonic"

https://github.com/nickrusso42518/py-auphonic

Python requests

Recap and Q&A

What We Learned

- HTTP/REST fundamentals
- Using requests to solve real business problems
- Minor options
  - Troubleshooting (logging, debugging)
  - Optimization (caching, session state)
  - Composition design with OOP

Additional Resources

- Website as code (extensive "requests" usage):
  - Videos: http://njrusmc.net/video/video.html
  - Process: http://njrusmc.net/ci_scripts/webcicd.html
- More Python Training by Me
  - O'Reilly: Learning Python 3 By Example
  - Pluralsight: Automating Networks with Python
- Twitter: @nickrusso42518