Principles/Social Media Mining

CIS 600/CPS 688

Week 5: Twitter APIs, Part 2: REST API

Edmund Yu, PhD
Associate Teaching Professor
esyu@syr.edu

September 22, 2020

Exam 1

Date: October 1, 2020, 9:30am-10:40am

Format:

Online; open book; ~20 questions (multiple-choice, fill-in-the-blank, short-answer/program)

Coverage:

1. All my slides up to and including those for the September 29 class
2. Python Basics

Note: You must log onto Blackboard to take the exam. You could also enter our Blackboard Course Room to ask me whatever question you may have about the exam. You are allowed to use your laptop to read my slides, our textbooks and your notes, but not for anything else. No other electronic devices are allowed. Communication with other people in any form is forbidden.

Twitter REST API

- Twitter **REST API** provides access to some core primitives of Twitter, including timelines (collections of tweets, ordered by time, most recent first), status updates, and user information

- Naturally, if you're building an application that leverages those Twitter objects (i.e. timelines, status updates, and user information), this is the API for you

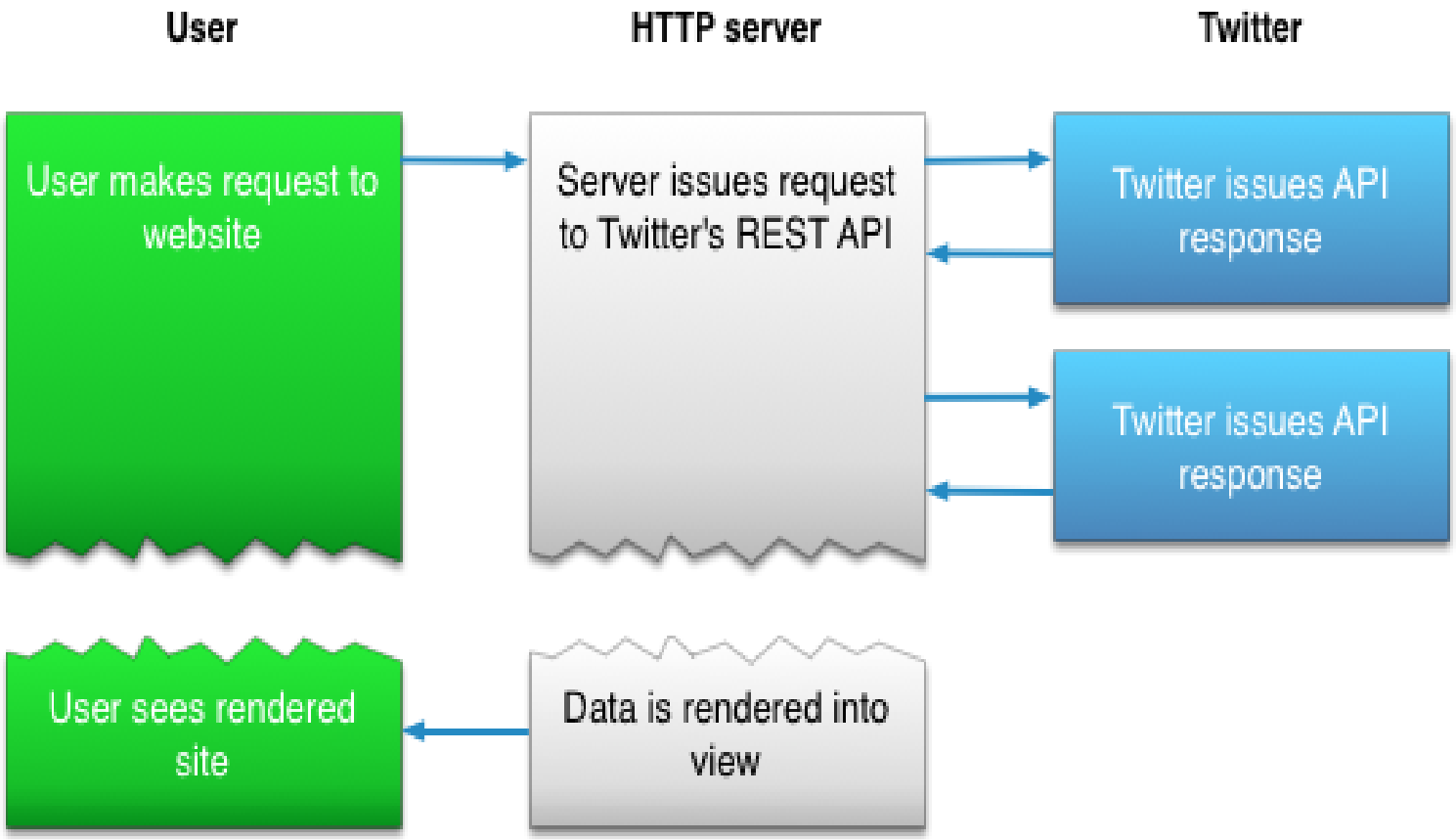
- In this course, we will use the REST API to harvest user information, especially the friends/followers information, in order to construct social networks for experimentation.

  → Assignment #2

- Official documentation at: **https://developer.twitter.com/en/docs/twitter-api/v1/accounts-and-users/follow-search-get-users/overview#**

REST

- REST (REpresentational State Transfer) is an <u>client-server</u> architectural style/pattern made popular by Roy Fielding's PhD dissertation



REST

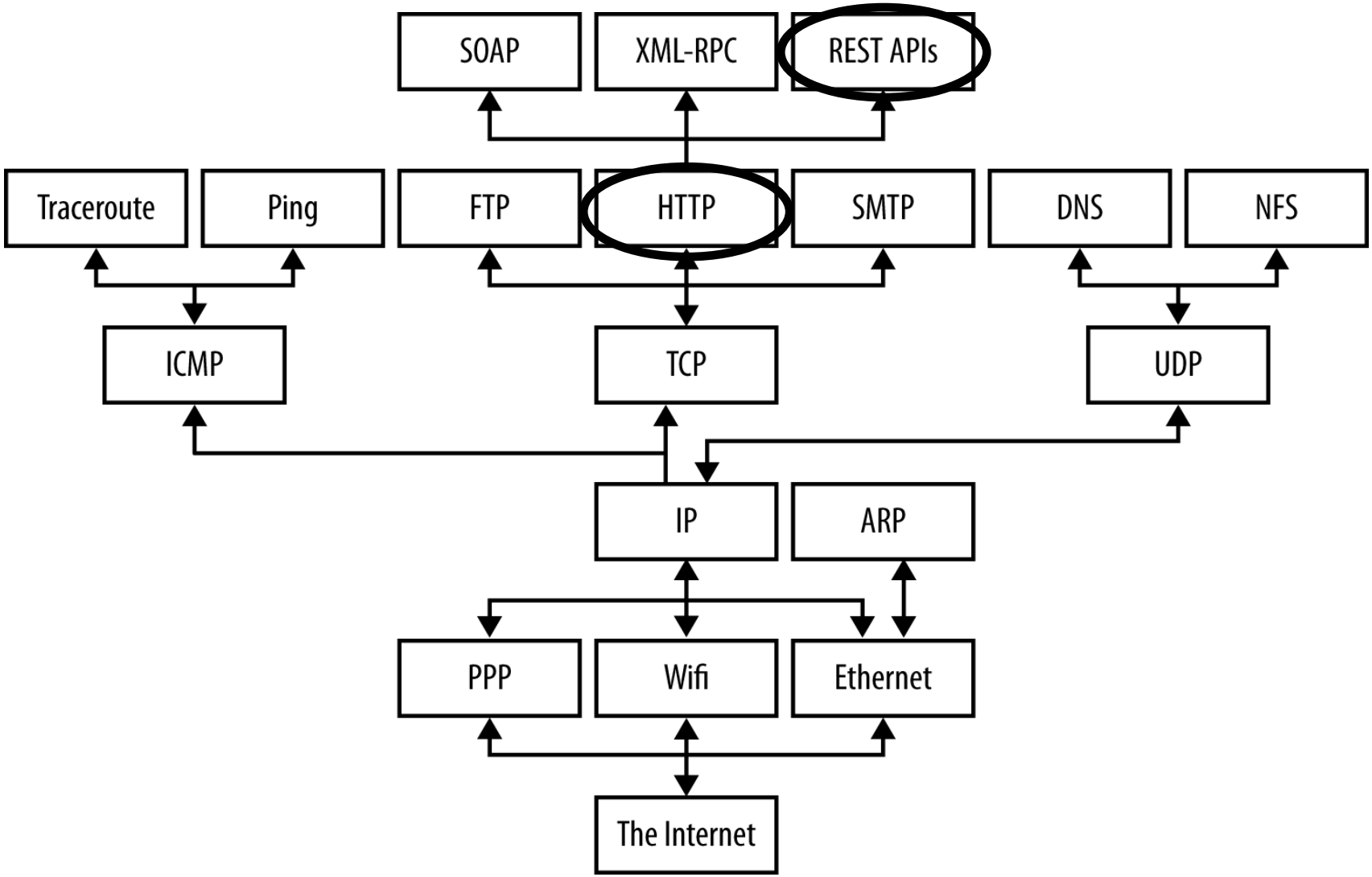
- Key principles:
  - Clients and servers should engage in stateless communication
  - URIs should describe resources that can be acted upon by standard HTTP verbs such as GET, PUT, POST, and DELETE. For example:

    http://example.com/tweet might describe a "tweet" resource

    - A **GET** on that resource would fetch it
    - A **PUT** on the context */tweet/foo* might create a new tweet for user "foo"
    - A **DELETE** on the context */tweet/foo* would delete the tweet.
    - A **POST** /tweet/foo might append a new tweet to foo's tweets.

- http://en.wikipedia.org/wiki/Representational_State_Transfer

Internet Protocols



Twitter API v1.1

Fundamentals
Tweets
Users
Subscribe to account activity
Manage account settings and profile
Mute, block, and report users
Follow, search, and get users
Create and manage lists
User profile images and banners
Direct Messages
Media
Trends
Geo
Metrics

Follow, search, and get users

Overview API reference

Please note:

We launched a new version of the standard users/lookup and users/show endpoints as part of Twitter API v2: Early Access. If you are currently using any of these endpoints, you can use our migration materials to start working with the newer endpoint.

Overview

The following API endpoints can be used to programmatically follow users, search for users, and get user information:

Friends and followers	POST friendships	Get user info
• GET followers/ids	• POST friendships/create	• GET users/lookup
• GET followers/list	• POST friendships/destroy	• GET users/search

Twitter API v1.1

Fundamentals
Tweets
Users
Subscribe to account activity
Manage account settings and profile
Mute, block, and report users
Follow, search, and get users
Create and manage lists
User profile images and banners
Direct Messages
Media
Trends
Geo
Metrics

Overview

The following API endpoints can be used to programmatically follow users, search for users, and get user information:

Friends and followers	POST friendships	Get user info
• GET followers/ids	• POST friendships/create	• GET users/lookup
• GET followers/list	• POST friendships/destroy	• GET users/search
• GET friends/ids	• POST friendships/update	• GET users/show
• GET friends/list		
• GET friendships/incoming		
• GET friendships/lookup		
• GET friendships/no_retweets/ids		
• GET friendships/outgoing		
• GET friendships/show		

For more details, please see the individual endpoint information within the API reference section.

Terminology

To avoid confusion around the term "friends" and "followers" with respect to the API endpoints, below is a definition of each:

Friends - we refer to "friends" as the Twitter users that a specific user follows (e.g., following). Therefore, the `GET friends/ids` endpoint returns a collection of user IDs that the specified user follows.

Documentation

Search the docs

Twitter API

Getting started
Tutorials
Tools and libraries
Migrate
API reference Index

Twitter API v2 Early Access

Fundamentals
Tweets
Users

GET users/show

Returns a variety of information about the user specified by the required `user_id` or `screen_name` parameter. The author's most recent Tweet will be returned inline when possible.

GET users / lookup is used to retrieve a bulk collection of user objects.

You must be following a protected user to be able to see their most recent Tweet. If you don't follow a protected user, the user's Tweet will be removed. A Tweet will not always be returned in the current_status field.

Resource URL

`https://api.twitter.com/1.1/users/show.json`

Resource Information

Response formats	JSON
Requires authentication?	Yes
Rate limited?	Yes
Requests / 15-min window (user auth)	900
Requests / 15-min window (app auth)	900

Twitter API v1.1
Fundamentals
Tweets
Users
Subscribe to account activity
Manage account settings and profile
Mute, block, and report users
Follow, search, and get users
Create and manage lists
User profile images and banners
Direct Messages
Media
Trends
Geo

Parameters

Name	Required	Description	Default Value	Example
user_id	required	The ID of the user for whom to return results. Either an id or screen_name is required for this method.		*12345*
screen_name	required	The screen name of the user for whom to return results. Either a id or screen_name is required for this method.		*noradio*
include_entities	optional	The *entities* node will not be included when set to *false*.		*false*

Example Request

```
$ curl --request GET
   --url 'https://api.twitter.com/1.1/users/show.json?screen_name=twitterdev'
   --header 'authorization: Bearer <bearer>'
$ curl --request GET
 --url 'https://api.twitter.com/1.1/users/show.json?screen_name=twitterdev'
 --header 'authorization: OAuth oauth_consumer_key="consumer-key-for-app",
 oauth_nonce="generated-nonce", oauth_signature="generated-signature",
 oauth_signature_method="HMAC-SHA1", oauth_timestamp="generated-timestamp",
 oauth_version="1.0"'
$ twurl /1.1/users/show.json?screen_name=twitterdev
```

Example Response

Fetching User Info

```
auth = twitter.oauth.OAuth(OAUTH_TOKEN, OAUTH_TOKEN_SECRET,
                           CONSUMER_KEY, CONSUMER_SECRET)
twitter_api = twitter.Twitter(auth=auth)
# or
twitter_api = oauth_login()  // Chapter 9, Twitter Cookbook

screen_name = ‘katyperry'
response = twitter_api.users.show(screen_name=screen_name)
print(json.dumps(response, sort_keys=True, indent=1))
```

GET users/show (https://developer.twitter.com/en/docs/twitter-api/v1/accounts-and-users/follow-search-get-users/api-reference/get-users-show)

- Returns a variety of information about the user specified by the required user_id or screen_name parameter. The author's most recent Tweet will be returned inline when possible.

Twitter API v1.1

Fundamentals
Tweets
Users
Subscribe to account activity
Manage account settings and profile
Mute, block, and report users
Follow, search, and get users
Create and manage lists
User profile images and banners
Direct Messages
Media
Trends
Geo
Metrics

Overview

The following API endpoints can be used to programmatically follow users, search for users, and get user information:

Friends and followers	POST friendships	Get user info
• GET followers/ids	• POST friendships/create	• GET users/lookup
• GET followers/list	• POST friendships/destroy	• GET users/search
• GET friends/ids	• POST friendships/update	• GET users/show
• GET friends/list		
• GET friendships/incoming		
• GET friendships/lookup		
• GET friendships/no_retweets/ids		
• GET friendships/outgoing		
• GET friendships/show		

For more details, please see the individual endpoint information within the API reference section.

Terminology

To avoid confusion around the term "friends" and "followers" with respect to the API endpoints, below is a definition of each:

Friends - we refer to "friends" as the Twitter users that a specific user follows (e.g., following). Therefore, the `GET friends/ids` endpoint returns a collection of user IDs that the specified user follows.

GET users/search

- Provides a simple, relevance-based search interface to public user accounts on Twitter…
- Exact match searches are not supported.
- Only the first **1,000** matching results are available.

```
# GET users/search
response = twitter_api.users.search(q='Katy Perry')
print(json.dumps(response, sort_keys=True, indent=1))
```

Back to Fetching User Info

```
auth = twitter.oauth.OAuth(OAUTH_TOKEN, OAUTH_TOKEN_SECRET,
                           CONSUMER_KEY, CONSUMER_SECRET)
twitter_api = twitter.Twitter(auth=auth)
# or
twitter_api = oauth_login()  // Chapter 9, Twitter Cookbook

screen_name = ‘katyperry'
response = twitter_api.users.show(screen_name=screen_name)
print(json.dumps(response, sort_keys=True, indent=1))
```

GET users/show (https://developer.twitter.com/en/docs/twitter-api/v1/accounts-and-users/follow-search-get-users/api-reference/get-users-show)

- Returns a variety of information about the user specified by the required user_id or screen_name parameter. The author's most recent Tweet will be returned inline when possible.

Fetching User Info

{…
"**created_at**": "Fri Feb 20 23:45:56 +0000 2009",…
"**description**": "Love. Light.",…
"**followers_count**": 108410012,…
"**friends_count**": 223,
"**id**": 21447363,…
"**name**": "KATY PERRY",…
"**screen_name**": "katyperry",
"**status**": {…
"**created_at**": "Mon Feb 03 03:50:48 +0000 2020",
"**entities**": {…}
"**retweet_count**": 589,…
"**text**": "do you know the hotter the fire the purer the gold is... @ Aulani Resort, Oahu, Hawaii https://t.co/hY2pKjhWDh",
},
"**statuses_count**": 10236,…
}

Documentation

Search the docs

Twitter API

Getting started
Tutorials
Tools and libraries
Migrate
API reference Index

Twitter API v2 Early Access

Fundamentals
Tweets
Users

User object

The User object contains Twitter User account metadata that describes the Twitter User referenced. Users can author Tweets, Retweet, quote other Users Tweets, reply to Tweets, follow Users, be @mentioned in Tweets and can be grouped into lists.

Fetching User Info

```
auth = twitter.oauth.OAuth(OAUTH_TOKEN, OAUTH_TOKEN_SECRET,
                           CONSUMER_KEY, CONSUMER_SECRET)
twitter_api = twitter.Twitter(auth=auth)
# or
twitter_api = oauth_login()  // Chapter 9, Twitter Cookbook

screen_name = ‘katyperry'
response = twitter_api.users.show(screen_name=screen_name)
Print(json.dumps(response, sort_keys=True, indent=1))
```

GET users/show (https://developer.twitter.com/en/docs/twitter-api/v1/accounts-and-users/follow-search-get-users/api-reference/get-users-show)

- You must be following a protected user to be able to see their most recent Tweet. If you don't follow a protected user, the users Tweet will be removed. A Tweet will not always be returned in the current_status field.

Twitter API v1.1

Fundamentals
Data Dictionary
Enrichments
Rules and filtering
Rate limits
Tweets
Users
Direct Messages
Media
Trends
Geo
Metrics
Developer utilities

GET endpoints

The standard API rate limits described in this table refer to GET (read) endpoints. Note that endpoints not listed in the chart default to 15 requests per allotted user. All request windows are 15 minutes in length. These rate limits apply to the standard API endpoints only, does not apply to premium APIs.

Endpoint	Requests / window per user	Requests / window per app
GET account/verify_credentials	75	0
GET application/rate_limit_status	180	180
GET favorites/list	75	75
GET followers/ids	15	15
GET followers/list	15	15
GET friends/ids	15	15
GET friends/list	15	15
GET friendships/show	180	15
GET geo/id/:place_id	75	0
GET help/configuration	15	15
GET help/languages	15	15
GET help/privacy	15	15

GET lists/subscribers/show	15	15
GET lists/subscriptions	15	15
GET search/tweets	180	450
GET statuses/lookup	900	300
GET statuses/mentions_timeline	75	0
GET statuses/retweeters/ids	75	300
GET statuses/retweets_of_me	75	0
GET statuses/retweets/:id	75	300
GET statuses/show/:id	900	900
GET statuses/user_timeline	900	1500
GET trends/available	75	75
GET trends/closest	75	75
GET trends/place	75	75
GET users/lookup	900	300
GET users/search	900	0
GET users/show	900	900
GET users/suggestions	15	15

Twitter API v1.1

Fundamentals
Data Dictionary
Enrichments
Rules and filtering
Rate limits
Tweets
Users
Direct Messages
Media
Trends
Geo
Metrics
Developer utilities

GET statuses/lookup	900	300
GET statuses/mentions_timeline	75	0
GET statuses/retweeters/ids	75	300
GET statuses/retweets_of_me	75	0
GET statuses/retweets/:id	75	300
GET statuses/show/:id	900	900
GET statuses/user_timeline	900	1500
GET trends/available	75	75
GET trends/closest	75	75
GET trends/place	75	75
GET users/lookup	900	300
GET users/search	900	0
GET users/show	900	900
GET users/suggestions	15	15
GET users/suggestions/:slug	15	15
GET users/suggestions/:slug/members	15	15

GET users/lookup

- If you want to get information for more than 1 user, you should you **GET users/lookup** (see next slide),which processes up to 100 users at a time, instead of **GET users/show,** which processes 1 user at a time.

Twitter API v1.1

Fundamentals
Tweets
Users
Subscribe to account activity
Manage account settings and profile
Mute, block, and report users
Follow, search, and get users
Create and manage lists
User profile images and banners
Direct Messages
Media
Trends
Geo
Metrics

GET users/lookup

Returns fully-hydrated user objects for up to 100 users per request, as specified by comma-separated values passed to the `user_id` and/or `screen_name` parameters.

This method is especially useful when used in conjunction with collections of user IDs returned from GET friends / ids and GET followers / ids.

GET users / show is used to retrieve a single user object.

There are a few things to note when using this method.

- You must be following a protected user to be able to see their most recent status update. If you don't follow a protected user their status will be removed.
- The order of user IDs or screen names may not match the order of users in the returned array.
- If a requested user is unknown, suspended, or deleted, then that user will not be returned in the results list.
- If none of your lookup criteria can be satisfied by returning a user object, a HTTP 404 will be thrown.
- You are strongly encouraged to use a POST for larger requests.

Resource URL

`https://api.twitter.com/1.1/users/lookup.json`

Resource Information

Response formats	JSON

Fetching User Info Using Lookup

```
auth = twitter.oauth.OAuth(OAUTH_TOKEN, OAUTH_TOKEN_SECRET,
                           CONSUMER_KEY, CONSUMER_SECRET)
twitter_api = twitter.Twitter(auth=auth)
# or twitter_api = oauth_login()

screen_names = 'katyperry, justinbieber' # a string, not a list
response = twitter_api.users.lookup(screen_name=screen_names)
Print(json.dumps(response, sort_keys=True, indent=1))
```

GET users/lookup (https://dev.twitter.com/rest/reference/get/users/lookup)

- Returns fully-hydrated user objects for up to 100 users per request, as specified by **comma-separated values** passed to the user_id and/or screen_name parameters.
- This method is especially useful when used in conjunction with collections of user IDs returned from GET friends/ids and GET followers/ids (later)

9.17. Resolving User Profile Information

9.17.1. Problem

You'd like to look up profile information for one or more user IDs or screen names.

9.17.2. Solution

Use the `GET users/lookup` API to exchange as many as 100 IDs or usernames at a time for complete user profiles.

9.17.3. Discussion

Many APIs, such as `GET friends/ids` and `GET followers/ids`, return opaque ID values that need to be resolved to usernames or other profile information for meaningful analysis. Twitter provides a `GET users/lookup` API that can be used to resolve as many as 100 IDs or usernames at a time, and a simple pattern can be employed to iterate over larger batches. Although it adds a little bit of complexity to the logic, a single function can be constructed that accepts keyword parameters for your choice of either usernames or IDs that are resolved to user profiles. Example 9-17 illustrates such a function that can be adapted for a large variety of purposes, providing ancillary support for situations in which you'll need to resolve user IDs.

9.16: Making Robust Twitter Requests

Problem

- You want to write a long-running script that harvests large amounts of data, such as the friend and follower ids for a very popular Twitterer; however, the Twitter API is inherently unreliable and imposes rate limits that require you to always expect the unexpected.

Solution

- Write an abstraction for making twitter requests that accounts for rate limiting and other types of HTTP errors so that you can focus on the problem at hand and not worry about HTTP errors or rate limits.

9.16: Making Robust Twitter Requests

```
def make_twitter_request(twitter_api_func, max_errors=10, *args, **kw):
    # A nested helper function that handles common HTTPErrors. Return an updated
    # value for wait_period if the problem is a 500 level error. Block until the
    # rate limit is reset if it's a rate limiting issue (429 error). Returns None
    # for 401 and 404 errors, which requires special handling by the caller.
    def handle_twitter_http_error(e, wait_period=2, sleep_when_rate_limited=True):
        if wait_period > 3600: # Seconds
            print('Too many retries. Quitting.', file=sys.stderr)
            raise e
        # See https://developer.twitter.com/en/docs/basics/response-codes
        # for common codes
        if e.e.code == 401:
            print('Encountered 401 Error (Not Authorized)', file=sys.stderr)
            return None
        elif e.e.code == 404:
            print('Encountered 404 Error (Not Found)', file=sys.stderr)
            return None
        …
```

See Chapter 9: Twitter Cookbook for complete code listing (on Blackboard)

List of HTTP status codes

From Wikipedia, the free encyclopedia

This is a list of Hypertext Transfer Protocol (HTTP) response status codes. Status codes are issued by a server in response to a client's request made to the server. It includes codes from IETF Request for Comments (RFCs), other specifications, and some additional codes used in some common applications of the HTTP. The first digit of the status code specifies one of five standard classes of responses. The message phrases shown are typical, but any human-readable alternative may be provided. Unless otherwise stated, the status code is part of the HTTP/1.1 standard (RFC 7231).[1]

The Internet Assigned Numbers Authority (IANA) maintains the official registry of HTTP status codes.[2]

All HTTP response status codes are separated into five classes or categories. The first digit of the status code defines the class of response, while the last two digits do not have any classifying or categorization role. There are five classes defined by the standard:

- *1xx informational response* – the request was received, continuing process
- *2xx successful* – the request was successfully received, understood, and accepted
- *3xx redirection* – further action needs to be taken in order to complete the request
- *4xx client error* – the request contains bad syntax or cannot be fulfilled
- *5xx server error* – the server failed to fulfil an apparently valid request

Contents [hide]

1 1xx informational response
2 2xx success
3 3xx redirection
4 4xx client errors
5 5xx server errors
6 Unofficial codes

HTTP

Persistence · Compression · HTTPS · QUIC

Request methods
OPTIONS · GET · HEAD · POST · PUT · DELETE · TRACE · CONNECT · PATCH

Header fields
Cookie · ETag · Location · HTTP referer · DNT · X-Forwarded-For

Status codes
301 Moved Permanently · 302 Found · 303 See Other · 403 Forbidden · 404 Not Found · 451 Unavailable for Legal Reasons

Security access control methods
Basic access authentication · Digest access authentication

Security vulnerabilities
HTTP header injection · HTTP request smuggling · HTTP response splitting · HTTP parameter pollution

V · T · E

Twitter Cookbook 9.17

```
def get_user_profile(twitter_api, screen_names=None, user_ids=None):
    # Must have either screen_name or user_id (logical xor)
    assert (screen_names != None) != (user_ids != None), \
    "Must have screen_names or user_ids, but not both"
    items_to_info = {}
    items = screen_names or user_ids
    while len(items) > 0:
        # Process 100 items at a time per the API specifications for /users/lookup.
        # See http://bit.ly/2Gcjfzr for details.
        items_str = ','.join([str(item) for item in items[:100]])
        items = items[100:]
        if screen_names:
            response = make_twitter_request(twitter_api.users.lookup, screen_name=items_str)
        else: # user_ids
            response = make_twitter_request(twitter_api.users.lookup, user_id=items_str)
```

See Chapter 9: Twitter Cookbook for complete code listing.

Twitter Cookbook 9.17

```
for user_info in response:
        if screen_names:
            items_to_info[user_info['screen_name']] = user_info
        else: # user_ids
            items_to_info[user_info['id']] = user_info

    return items_to_info

# Sample usage

twitter_api = oauth_login()

print(get_user_profile(twitter_api, screen_names=["SocialWebMining", "ptwobrussell"]))
#print(get_user_profile(twitter_api, user_ids=[132373965]))
```

GET statuses/retweets_of_me	statuses	75	0
GET statuses/retweets/:id	statuses	75	300
GET statuses/show/:id	statuses	900	900
GET statuses/user_timeline	statuses	900	1500
GET trends/available	trends	75	75
GET trends/closest	trends	75	75
GET trends/place	trends	75	75
GET users/lookup	users	900	300
GET users/search	users	900	0
GET users/show	users	900	900
GET users/suggestions	users	15	15
GET users/suggestions/:slug	users	15	15
GET users/suggestions/:slug/members	users	15	15

Twitter API v1.1

Fundamentals

Tweets

Users

Subscribe to account activity

Manage account settings and profile

Mute, block, and report users

Follow, search, and get users

Create and manage lists

User profile images and banners

Direct Messages

Media

Trends

Geo

Metrics

GET friends/ids

Returns a cursored collection of user IDs for every user the specified user is following (otherwise known as their "friends").

At this time, results are ordered with the most recent following first — however, this ordering is subject to unannounced change and eventual consistency issues. Results are given in groups of 5,000 user IDs and multiple "pages" of results can be navigated through using the `next_cursor` value in subsequent requests. See Using cursors to navigate collections for more information.

This method is especially powerful when used in conjunction with GET users / lookup, a method that allows you to convert user IDs into full user objects in bulk.

Resource URL

`https://api.twitter.com/1.1/friends/ids.json`

Resource Information

Response formats	JSON
Requires authentication?	Yes
Rate limited?	Yes
Requests / 15-min window (user auth)	15
Requests / 15-min window (app auth)	15

Twitter API v1.1

Fundamentals

Tweets

Users

Subscribe to account activity

Manage account settings and profile

Mute, block, and report users

Follow, search, and get users

Create and manage lists

User profile images and banners

Direct Messages

Media

Trends

Geo

Metrics

Parameters

Name	Required	Description	Default Value	Example
user_id	optional	The ID of the user for whom to return results.		`12345`
screen_name	optional	The screen name of the user for whom to return results.		`noradio`
cursor	semi-optional	Causes the list of connections to be broken into pages of no more than 5000 IDs at a time. The number of IDs returned is not guaranteed to be 5000 as suspended users are filtered out after connections are queried. If no cursor is provided, a value of `-1` will be assumed, which is the first "page." The response from the API will include a `previous_cursor` and `next_cursor` to allow paging back and forth. See Using cursors to navigate collections for more information.	`-1`	`1289376451093 8`
stringify_ids	optional	Some programming environments will not consume Twitter IDs due to their size. Provide this option to have IDs returned as strings instead. More about Twitter IDs.	`false`	`true`
count	optional	Specifies the number of IDs attempt retrieval of, up to a maximum of 5,000 per distinct request. The value of `count` is best thought of as a limit to the number of		`2048`

Twitter API v1.1

Fundamentals
Tweets
Users
Subscribe to account activity
Manage account settings and profile
Mute, block, and report users
Follow, search, and get users
Create and manage lists
User profile images and banners
Direct Messages
Media
Trends
Geo
Metrics

GET followers/ids

Returns a cursored collection of user IDs for every user following the specified user.

At this time, results are ordered with the most recent following first — however, this ordering is subject to unannounced change and eventual consistency issues. Results are given in groups of 5,000 user IDs and multiple "pages" of results can be navigated through using the `next_cursor` value in subsequent requests. See Using cursors to navigate collections for more information.

This method is especially powerful when used in conjunction with GET users / lookup, a method that allows you to convert user IDs into full user objects in bulk.

Resource URL

`https://api.twitter.com/1.1/followers/ids.json`

Resource Information

Response formats	JSON
Requires authentication?	Yes
Rate limited?	Yes
Requests / 15-min window (user auth)	15
Requests / 15-min window (app auth)	15

Twitter API v1.1

Fundamentals

Tweets

Users

Subscribe to account activity

Manage account settings and profile

Mute, block, and report users

Follow, search, and get users

Create and manage lists

User profile images and banners

Direct Messages

Media

Trends

Geo

Metrics

Parameters

Name	Required	Description	Default Value	Example
user_id	optional	The ID of the user for whom to return results.		`12345`
screen_name	optional	The screen name of the user for whom to return results.		`noradio`
cursor	semi-optional	Causes the list of connections to be broken into pages of no more than 5000 IDs at a time. The number of IDs returned is not guaranteed to be 5000 as suspended users are filtered out after connections are queried. If no cursor is provided, a value of `-1` will be assumed, which is the first "page." The response from the API will include a `previous_cursor` and `next_cursor` to allow paging back and forth. See Using cursors to navigate collections for more information.	`-1`	`1289376451093 8`
stringify_ids	optional	Some programming environments will not consume Twitter IDs due to their size. Provide this option to have IDs returned as strings instead. More about Twitter IDs.	`false`	`true`
count	optional	Specifies the number of IDs attempt retrieval of, up to a maximum of 5,000 per distinct request. The value of `count` is best thought of as a limit to the number of		`2048`

Get Friends

```
import twitter
import json

twitter_api = oauth_login()
screen_name = 'katyperry'
response = twitter_api.friends.ids(screen_name=screen_name, count=5000)
print (json.dumps(response, indent=1,sort_keys=True))

friends = response["ids"]
print('got {0} friends for {1}'.format(len(friends), screen_name))
# See next slide for ‘format()’
```

format() (Review)

- format(...)

  S.format(*args, **kwargs) → string

- Return a formatted version of S, using substitutions from args and kwargs.

- The substitutions are identified by braces ('{' and '}').

Get Followers

```
import twitter
import json

twitter_api = oauth_login()
screen_name = 'katyperry'
response = twitter_api.followers.ids(screen_name=screen_name, count = 5000)
print(json.dumps(response, indent=1))

followers = response["ids"]
print ('got {0} followers for {1}'.format(len(followers), screen_name))
```

Using make_twitter_request()

response = **twitter_api.friends.ids(screen_name=screen_name, count = 5000)**

response = **make_twitter_request**(twitter_api.friends.ids,…)

Get More Friends/Followers

```
cursor = response["next_cursor"]

if (cursor != 0):
    response = twitter_api.friends.ids(screen_name=screen_name,
            count = 5000, cursor=cursor)
    friends += response["ids"]
```

Get More Friends/Followers

```
for i in range(10):
    cursor = response["next_cursor"]
    if (cursor != 0):
        response = twitter_api.friends.ids(screen_name=screen_name,
                    count = 5000,  cursor=cursor)
        friends += response["ids"]
```

9.19: Getting All Friends/Followers

Problem

You'd like to harvest all of the friends or followers for a (potentially very popular) Twitter user.

Solution

Use the **make_twitter_request** function introduced in Section 9.16 on page 377 to simplify the process of harvesting IDs by accounting for situations in which the number of followers may exceed what can be fetched within the prescribed rate limits.

9.19: Getting All Friends/Followers

Discussions

- The **GET followers/ids** and **GET friends/ids** provide an API that can be navigated to retrieve all of the follower and friend IDs for a particular user, but the logic involved in retrieving all of the IDs can be nontrivial since each API request returns at most **5,000** IDs at a time.
- Popular users or **celebrities** that are more interesting to analyze often have hundreds of thousands or even millions of followers.
- Harvesting all of these IDs can be challenging because of the need to walk the **cursor** for each batch of results and also account for possible HTTP errors along the way.
- Fortunately, it's not too difficult to adapt **make_twitter_request** and previously introduced logic for walking the cursor of results to systematically fetch all of these ids.
- It is advisable to store the results into a document-oriented database, such as **MongoDB**, after each result so that no information is ever lost in the event of an unexpected glitch during a large harvesting operation.

9.19: Getting All Friends/Followers

```
from functools import partial
from sys import maxsize as maxint

def get_friends_followers_ids(twitter_api, screen_name=None, user_id=None,
                              friends_limit=maxint, followers_limit=maxint):
    # Must have either screen_name or user_id (logical xor)
    assert (screen_name != None) != (user_id != None), \
    "Must have screen_name or user_id, but not both"
    # See http://bit.ly/2GcjKJP and http://bit.ly/2rFz90N for details
    # on API parameters
    get_friends_ids = partial(make_twitter_request, twitter_api.friends.ids,
                              count=5000)
    get_followers_ids = partial(make_twitter_request, twitter_api.followers.ids,
                                count=5000)
    friends_ids, followers_ids = [], []
    …
```

See Chapter 9: Twitter Cookbook for complete code listing

9.5: Constructing Convenient Function Calls

Problem

You want to bind certain parameters to function calls and pass around a reference to the bound function in order to simplify coding.

Solution

Use Python's **functools.partial** to create fully or partially bound functions that can be elegantly passed around and invoked by other code without the need to pass additional parameters.

9.5: Constructing Convenient Function Calls

- **functools.partial** is a incredibly convenient tool to use in combination with the twitter package and many of the patterns in this cookbook, and in other Python programming books and examples.

- You may find it cumbersome to continually pass around a reference to an authenticated Twitter API object (twitter_api), which is usually the first argument to most functions.
  - To solve that problem, you could simply create a function that **partially** satisfies the function arguments so that you can freely pass around a function that can be invoked with its remaining parameters later. (Examples on next slide)

9.5: Constructing Convenient Function Calls

```
from functools import partial

pp = partial(json.dumps, indent=1)
twitter_world_trends = partial(twitter_trends, twitter_api, WORLD_WOE_ID)

print(pp(twitter_world_trends()))


authenticated_twitter_search = partial(twitter_search, twitter_api)
results = authenticated_twitter_search("iPhone")
print(pp(results))


authenticated_iphone_twitter_search = partial(authenticated_twitter_search,
"iPhone")
results = authenticated_iphone_twitter_search()
print(pp(results))
```

Administrator: Command Prompt - python -mpydoc functools

```
Microsoft Windows [Version 10.0.18362.592]
(c) 2019 Microsoft Corporation. All rights reserved.

C:¥WINDOWS¥system32>python -mpydoc functools
Help on module functools:

NAME
    functools - functools.py - Tools for working with functions and callable objects

CLASSES
    builtins.object
        partial
        partialmethod

    class partial(builtins.object)
     |  partial(func, *args, **keywords) - new function with partial application
     |  of the given arguments and keywords.
     |
     |  Methods defined here:
     |
     |  __call__(self, /, *args, **kwargs)
     |      Call self as a function.
     |
     |  __delattr__(self, name, /)
     |      Implement delattr(self, name).
     |
     |  __getattribute__(self, name, /)
     |      Return getattr(self, name).
     |
```

Administrator: Command Prompt - python -mpydoc functools.partial

```
C:¥WINDOWS¥system32>python -mpydoc functools.partial
Help on class partial in functools:

functools.partial = class partial(builtins.object)
 |  partial(func, *args, **keywords) - new function with partial application
 |  of the given arguments and keywords.
 |
 |  Methods defined here:
 |
 |  __call__(self, /, *args, **kwargs)
 |      Call self as a function.
 |
 |  __delattr__(self, name, /)
 |      Implement delattr(self, name).
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  __reduce__(...)
 |      helper for pickle
 |
 |  __repr__(self, /)
 |      Return repr(self).
```

Administrator: Command Prompt - python

```
>>> help(functools)
Help on module functools:

NAME
    functools - functools.py - Tools for working with functions and callable objects

CLASSES
    builtins.object
        partial
        partialmethod

    class partial(builtins.object)
     |  partial(func, *args, **keywords) - new function with partial application
     |  of the given arguments and keywords.
     |
     |  Methods defined here:
     |
     |  __call__(self, /, *args, **kwargs)
     |      Call self as a function.
     |
     |  __delattr__(self, name, /)
     |      Implement delattr(self, name).
     |
     |  __getattribute__(self, name, /)
     |      Return getattr(self, name).
     |
     |  __new__(*args, **kwargs) from builtins.type
     |      Create and return a new object.  See help(type) for accurate signature.
     |
     |  __reduce__(...)
     |      helper for pickle
     |
     |  __repr__(self, /)
     |      Return repr(self).
     |
     |  __setattr__(self, name, value, /)
-- More --
```

Administrator: Command Prompt - python

```
>>>
>>>
>>> help(functools.partial)
Help on class partial in module functools:

class partial(builtins.object)
 |  partial(func, *args, **keywords) - new function with partial application
 |  of the given arguments and keywords.
 |
 |  Methods defined here:
 |
 |  __call__(self, /, *args, **kwargs)
 |      Call self as a function.
 |
 |  __delattr__(self, name, /)
 |      Implement delattr(self, name).
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  __reduce__(...)
 |      helper for pickle
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __setattr__(self, name, value, /)
 |      Implement setattr(self, name, value).
 |
 |  __setstate__(...)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
```

Common Friends

```
twitter_api = oauth_login()
screen_name1= ‘katyperry‘
screen_name2= ‘justinbieber’

response = make_twitter_request(twitter_api.friends.ids,
    screen_name=screen_name1, count = 5000)
friends1 = response["ids"]

response = make_twitter_request(twitter_api.friends.ids,
    screen_name=screen_name2, count = 5000)
friends2= response["ids"]

common_friends = set(friends1) & set(friends2)
```

Reciprocal Friends

```
twitter_api = oauth_login()
screen_name = ‘katyperry’

response = make_twitter_request(twitter_api.friends.ids,
    screen_name=screen_name, count = 5000)
friends = response["ids"]


response = make_twitter_request(twitter_api.followers.ids,
    screen_name=screen_name, count = 5000)
followers = response["ids"]


reciprocal_friends = set(friends) & set(followers)
```

Get Friends of Friends

```
twitter_api = oauth_login()
screen_name = ‘katyperry’

response = make_twitter_request(twitter_api.friends.ids,
    screen_name=screen_name, count = 2)
ids = response[“ids”]
friends_of_friends = []

for id in ids:
    response = make_twitter_request(twitter_api.friends.ids,
        user_id=id, count = 2)
    friends_of_friends += response[“ids”]
```

9.22. Crawling a Friendship Graph

Problem

You'd like to harvest the IDs of a user's followers, followers of those followers, followers of followers of those followers, and so on, as part of a network analysis - essentially crawling a friendship graph of the "**following**" relationships on Twitter.

Solution

Use a breadth-first search to systematically harvest friendship information that can rather easily be interpreted as a graph for network analysis.

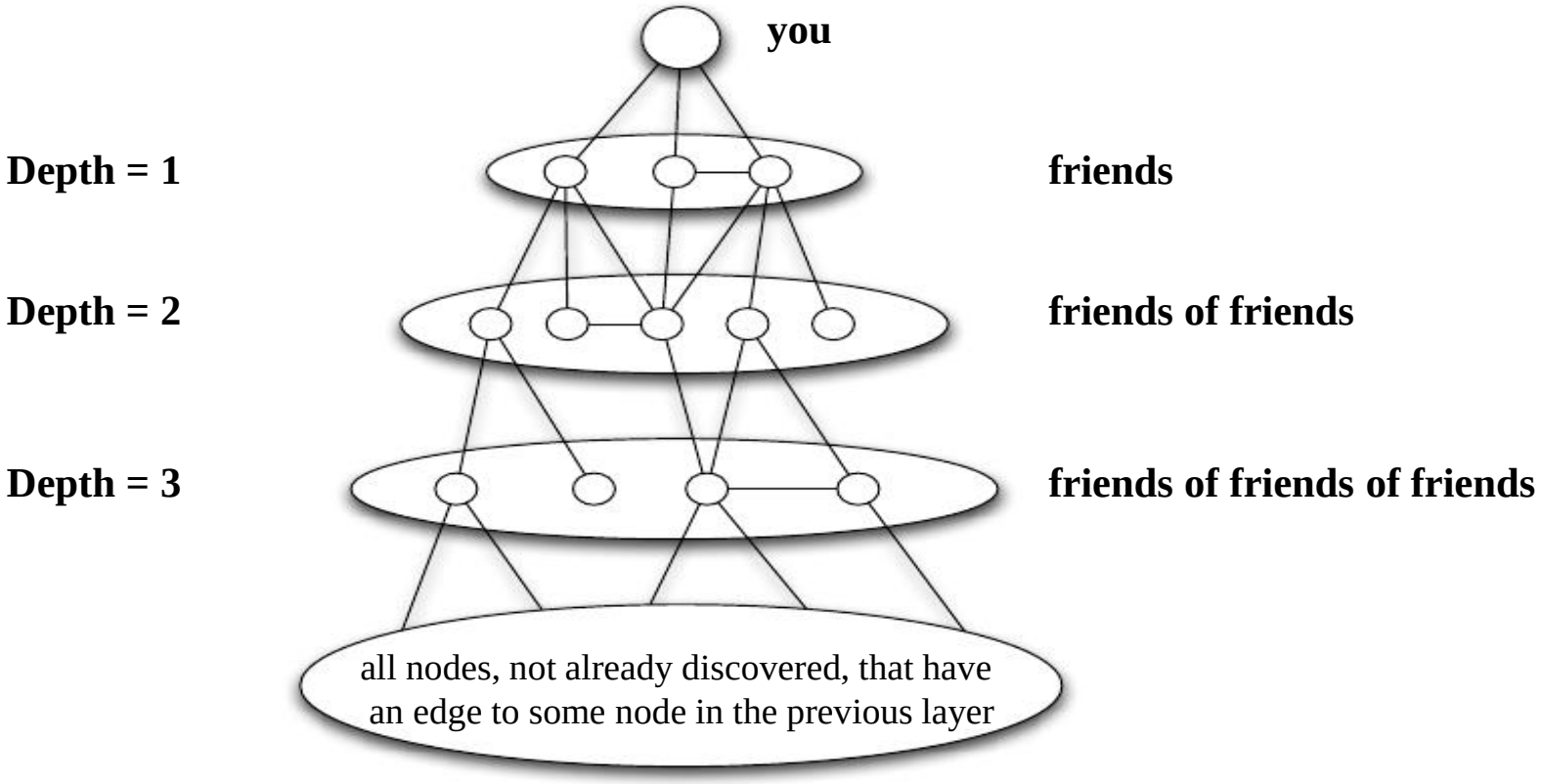
9.22. Crawling a Friendship Graph

Discussions

- A **breadth-first search** is a common technique for exploring a graph.
- Given a **starting point** and a **depth**, a breadth-first traversal systematically explores the space such that it is guaranteed to eventually return all nodes in the graph up to the said depth, and the search explores the space such that each depth completes before the next depth is begun (see Example 9-22).
- Keep in mind that it is quite possible that in exploring Twitter friendship graphs, you may encounter **supernodes -** nodes with very high degrees of outgoing edges – which can very easily consume computing resources and API requests that count toward your rate limit.
- It is advisable that you provide a meaningful cap on the maximum number of followers you'd like to fetch for each user in the graph, and determine whether the supernodes are worth the time and trouble.
- Exploring unknown graph is a complex (and exciting) problem to work on, and various other tools, such as sampling techniques, could be intelligently incorporated to further enhance the efficacy of the search.

Breadth-First Search

- For graphs that are a bit more complicated, we need a systematic method to determine distances.
- Breadth-first search



CB 9.22. Crawling a Friendship Graph

```
# **Example 22. Crawling a friendship graph**
def crawl_followers(twitter_api, screen_name, limit=1000000, depth=2,
**mongo_conn_kw):

    # Resolve the ID for screen_name and start working with IDs for consistency
    # in storage

    seed_id = str(twitter_api.users.show(screen_name=screen_name)['id'])

    _, next_queue = get_friends_followers_ids(twitter_api, user_id=seed_id,
                                              friends_limit=0, followers_limit=limit)

    # Store a seed_id => _follower_ids mapping in MongoDB

    save_to_mongo({'followers' : [ _id for _id in next_queue ]}, 'followers_crawl',
            '{0}-follower_ids'.format(seed_id), **mongo_conn_kw)
```

You may remove the statements involving MongoDB

9.22. Crawling a Friendship Graph

```
d = 1
while d < depth:
    d += 1
    (queue, next_queue) = (next_queue, [])
    for fid in queue:
        _, follower_ids = get_friends_followers_ids(twitter_api, user_id=fid,
                                                    friends_limit=0,
                                                    followers_limit=limit)

        # Store a fid => follower_ids mapping in MongoDB
        save_to_mongo({'followers' : [ _id for _id in follower_ids ]},
                      'followers_crawl', '{0}-follower_ids'.format(fid))

        next_queue += follower_ids
```

You may remove the statements involving MongoDB

A Simplified Crawler

```
# This is a simplified version of the crawler in the Cookbook
screen_name = ‘katyperry’
response = make_twitter_request(twitter_api.followers.ids, screen_name=screen_name, count=2)
ids = next_queue = response["ids"]
depth = 1
max_depth = 4

while depth  < max_depth:
    depth  += 1
    (queue, next_queue) = (next_queue, [])
    for id in queue:
        response = make_twitter_request(twitter_api.followers.ids, user_id=id, count = 2)
        next_queue += response[“ids”]
    ids += next_queue

print(ids)
```

Check out the Cookbook version of the Crawler on the next 2 slides.

RestAPI.py - C:\Users\Chris\Desktop\SMM-code\RestAPI.py (3.6.0)

File Edit Format Run Options Window Help

```
# Simplified Version of the crawler using make_twitter_request
screen_name = 'edmundyu1001'
response = make_twitter_request(twitter_api.followers.ids, screen_name=screen_name, count = 2)
next_queue = response["ids"]
ids = next_queue
print("Got followers for {0}: {1}".format(screen_name, ids))
depth = 1
max_depth = 4

while depth  < max_depth:
    depth  += 1
    (queue, next_queue) = (next_queue, [])

    for id in queue:
        response = make_twitter_request(twitter_api.followers.ids, user_id=id, count = 2)
        if response:
            print("Got followers for {0}: {1}".format(id, response["ids"]))
            for i in response["ids"]:
                if (i not in next_queue and i not in ids):
                    next_queue.append(i) # not in the cookbook version
            #next_queue += response["ids"] #not good enough
        else:
            print(str(id) + ' is protected')

    ids += next_queue

print(ids)
```

A Better Crawler

Ln: 153 Col: 42

Assignment #2 (10 points): Due Oct 4

1. Select a ‘starting point,’ i.e. a user on Twitter, which could be yourself or somebody else.

2. Retrieve his/her friends, which should be a list of id’s, and followers, which is another list of id’s, perhaps using the **get_friends_followers_id()** function from the Cookbook, or your own program if you prefer. Note: When you use get_friends_followers_id() or its equivalent, you are allowed to set the maximum number of friends and followers to be 5000 (but no less), in order to save API calls, and hence your time.

3. Use those 2 lists from Step 2 to find **reciprocal friends**, which is yet another list of id’s. (The definition of ‘reciprocal friends’ can be found in my slides.) These are the **distance-1 friends**.

Assignment #2 (10 points): Due Oct 4

4. From that list of reciprocal friends, select **5** most popular friends, as determined by their **followers_count** in their user profiles. (I suggest you use the **get_user_profile()** function from the Cookbook to retrieve the user profiles of the reciprocal friends.)
5. Repeat this process (Steps 2, 3 & 4) for each of the distance-1 friends, then distance-2 friends, so on and so forth, using a **crawler**, until you have gather at least **100** users/nodes for your social network. Note: I suggest you modify the crawler (**crawl_followers()**) function from the Cookbook or my simplied crawler to do this. However, please note that either one of these 2 crawlers retrieves only followers. You need to modify it to get both followers and friends, in order to compute the reciprocal friends .

Assignment #2 (10 points): Due Oct 4

6. Create a social network based on the results (nodes and edges) from Step 5, using the **Networkx** package, adding all the nodes and edges.

7. Calculate the **diameter** and **average distance** of your network, using certain built-in functions provided by Networkx (in 3.23 Distance Measures & 3.51 Shortest Paths, or your own functions if you prefer.

Assignment #2 (10 points): Due Oct 4

Deliverables

a) **Program output**: Your program should out output Network size, in terms of numbers of nodes & edges, average distance & diameter. Save program output to a file.

b) **Your program source code with comments describing each class, function or program segment**. Make sure it runs. Also indicate which part is your own code. Note: reusing code from the textbook/cookbook, my slides, and any python libs is allowed, but you should cite your source.)

c) Put your program output file, source code (with comments), and any data file in a folder, zip it and submit the zipped folder via Blackboard.

Grading Rubrics

- Program not running: -3
- Program crashed: -2 or -1 depending on when
- Fewer than 100 nodes collected: less than 90: -0.5; additional -0.5 per 10 nodes less than 90
- Diameter not (correctly) calculated: -1
- Average distance not (correctly) calculated: -1
- Network not created: -3
- Network not created correctly: -2
- Reciprocal friends not done correctly: -2
- Top 5 reciprocal friends not done correctly: -2
- No crawler: -3
- Crawling not done correctly: -2
- Low quality code, or no comments: -1
- Other unforeseen issues: depends on the severity