Algorithmic Methods of Data Mining

Graphs

Ioannis Chatzigiannakis

Sapienza University of Rome

Laboratory 9

Twitter API

- User-related functionality
  - Search, Followers, Friends, Send Messages ...
- Timeline-related functionality
  - Look-up, Search, Retweet ...
- Status-related functionality
  - Account-related functinality
- ...

oauth2 based access

- OAuth 2 is a method of authentication where an application makes API requests without the user context.
- Use this method if you just need read-only access to public information.
- Create Developer Account
- Register your Application
- Acquire Access Token
  - Consumer Key
  - Consumer Secret
  - Access Token
  - Access Token Secret

Tweepy Library

- An easy-to-use Python library for accessing the Twitter API.
- Provides access to the entire twitter RESTful API methods.
- Supports both OAuth 1a (application-user) and OAuth 2 (application-only) authentication.
- Supports Pagination.
- Supports Streaming mode for receiving results.

```
pip3 install tweepy
```

Getting Started: Authentication

```
import tweepy
from tweepy import OAuthHandler

cfg = {
    "consumer_key"        : "...",
    "consumer_secret"     : "...",
    "access_token"        : "...",
    "access_token_secret" : "..."
}

auth = OAuthHandler(cfg["consumer_key"],
                    cfg["consumer_secret"])
auth.set_access_token(cfg["access_token"],
                      cfg["access_token_secret"])

api = tweepy.API(auth)
```

User Lookup

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

```
user = api.get_user('ichatzi')
```

- Models contain the data and some helper methods which we can then use:

```
print(user.screen_name)
print(user.followers_count)

api.create_friendship(user.id)

for friend in user.followers():
  print(friend.screen_name)
```

User Data Dictionary

id	Int64	The unique identifier for this User.
id_str	String	The string representation of the unique identifier for this User.
name	String	The name of the user, as they've defined it.
screen_name	String	The screen name, handle, or alias that this user identifies themselves with.
location	String	Nullable. The user-defined location for this account's profile.
url	String	Nullable. A URL provided by the user in association with their profile.
description	String	Nullable. The user-defined UTF-8 string describing their account.
verified	Boolean	When true, indicates that the user has a verified account.

User Data Dictionary (3)

`followers_count`	`Int`	The number of followers this account currently has.
`friends_count`	`Int`	The number of users this account is following (AKA their "followings").
`listed_count`	`Int`	The number of public lists that this user is a member of.
`favourites_count`	`Int`	The number of Tweets this user has liked in the account's lifetime.
`statuses_count`	`Int`	The number of Tweets (including retweets) issued by the user.
`created_at`	`String`	UTC datetime of user creation.
`profile_banner_url`	`String`	URL pointing to the user's uploaded profile banner.
`profile_image_url_https`	`String`	URL pointing to the user's profile image.

Searching Users

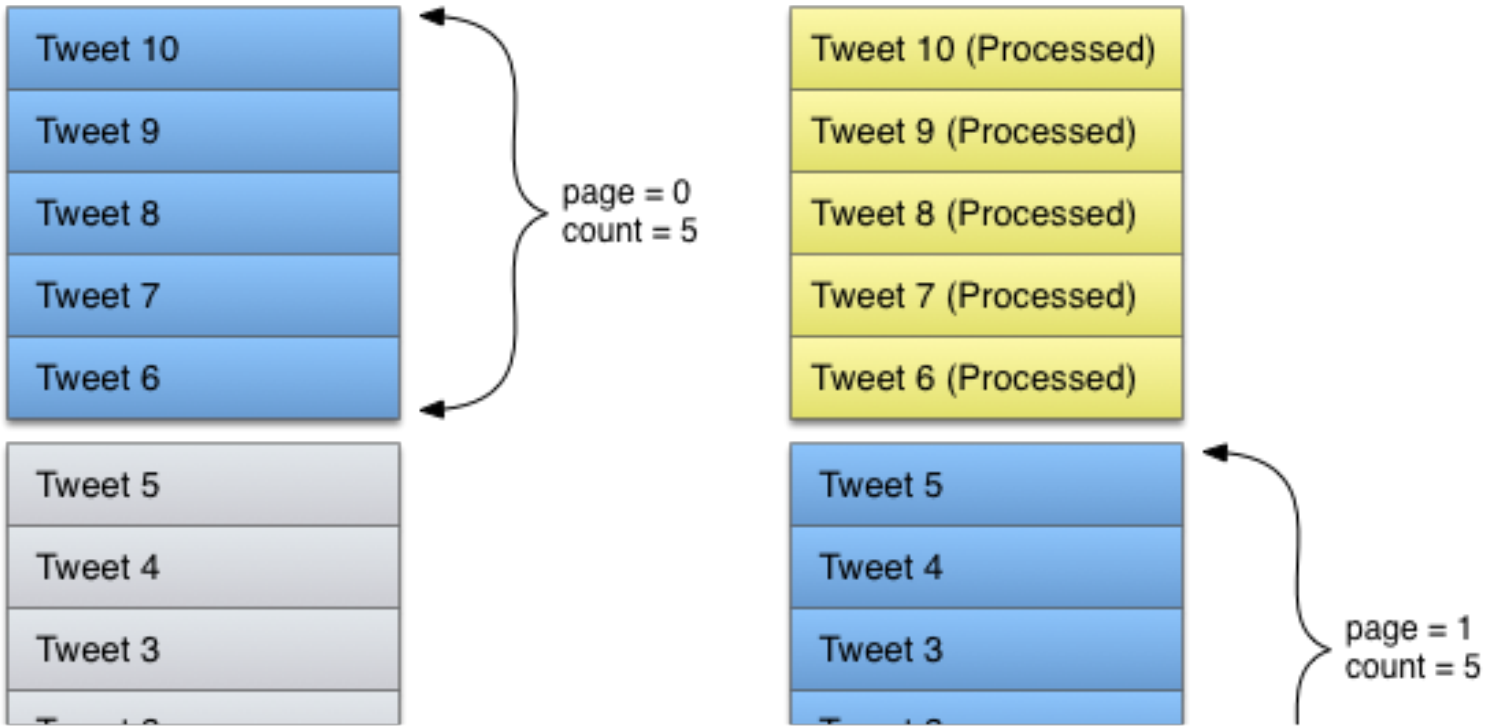
```
users = api.search_users("Roma", 10)
for user in users:
    print(user.screen_name)
```

Timeline Tweets

```
for status in api.user_timeline():
    # process status here
    print(status.text)
    print(status.user.screen_name)
    print(status.created_at, "Fav: ", status.favorite_count)
    print("---")
```

Retrieving Results

- Retrieves 10 reverse-chronologically sorted Tweets.
- page size = 5 elements and requesting the first page.

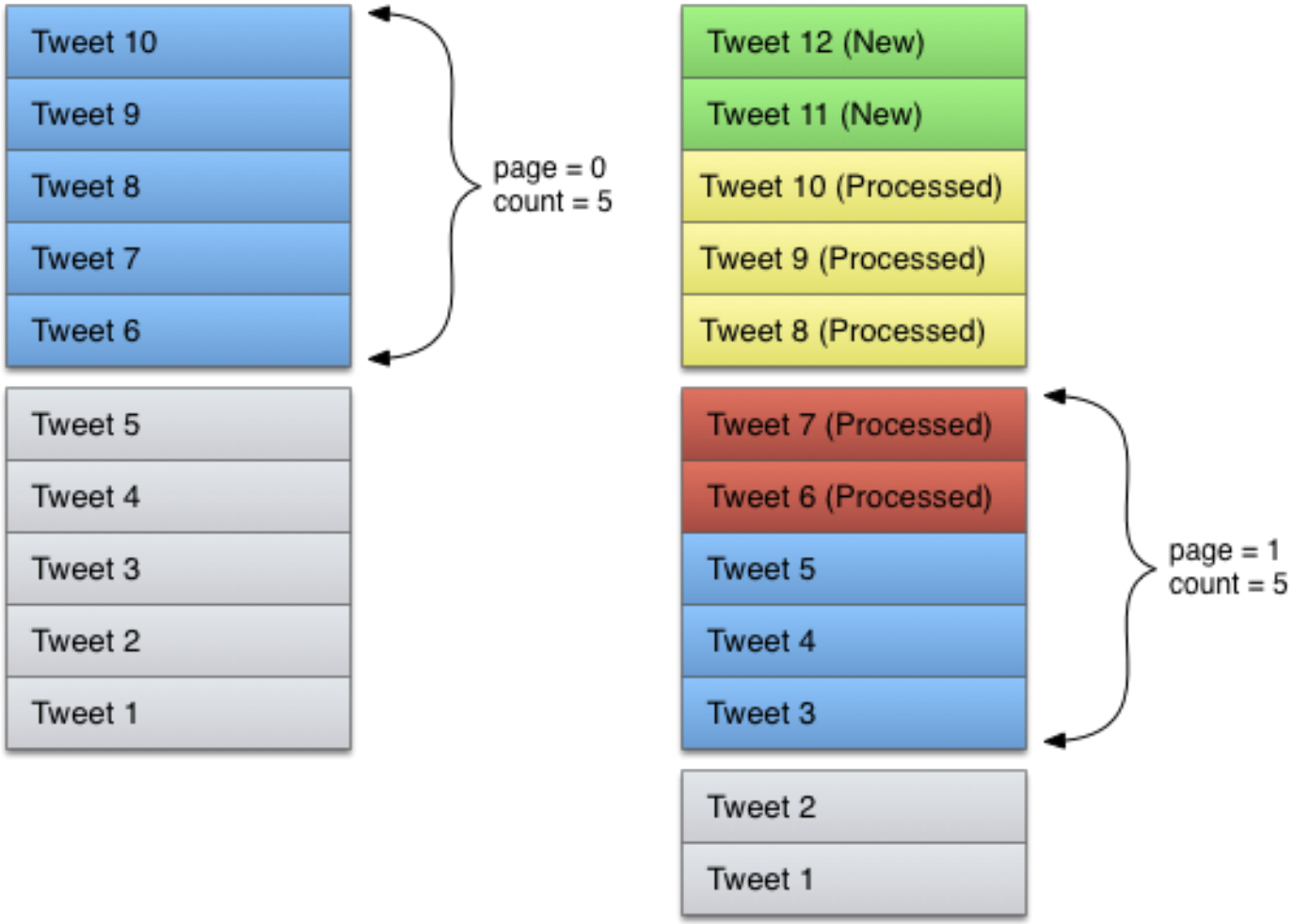


Retrieving Timeline Tweets with Pagination

```
page = 1
while True:
    statuses = api.user_timeline(page=page)
    if statuses:
        for status in statuses:
            # process status here
            process_status(status)
    else:
        # All done
        break
    page += 1  # next page
```

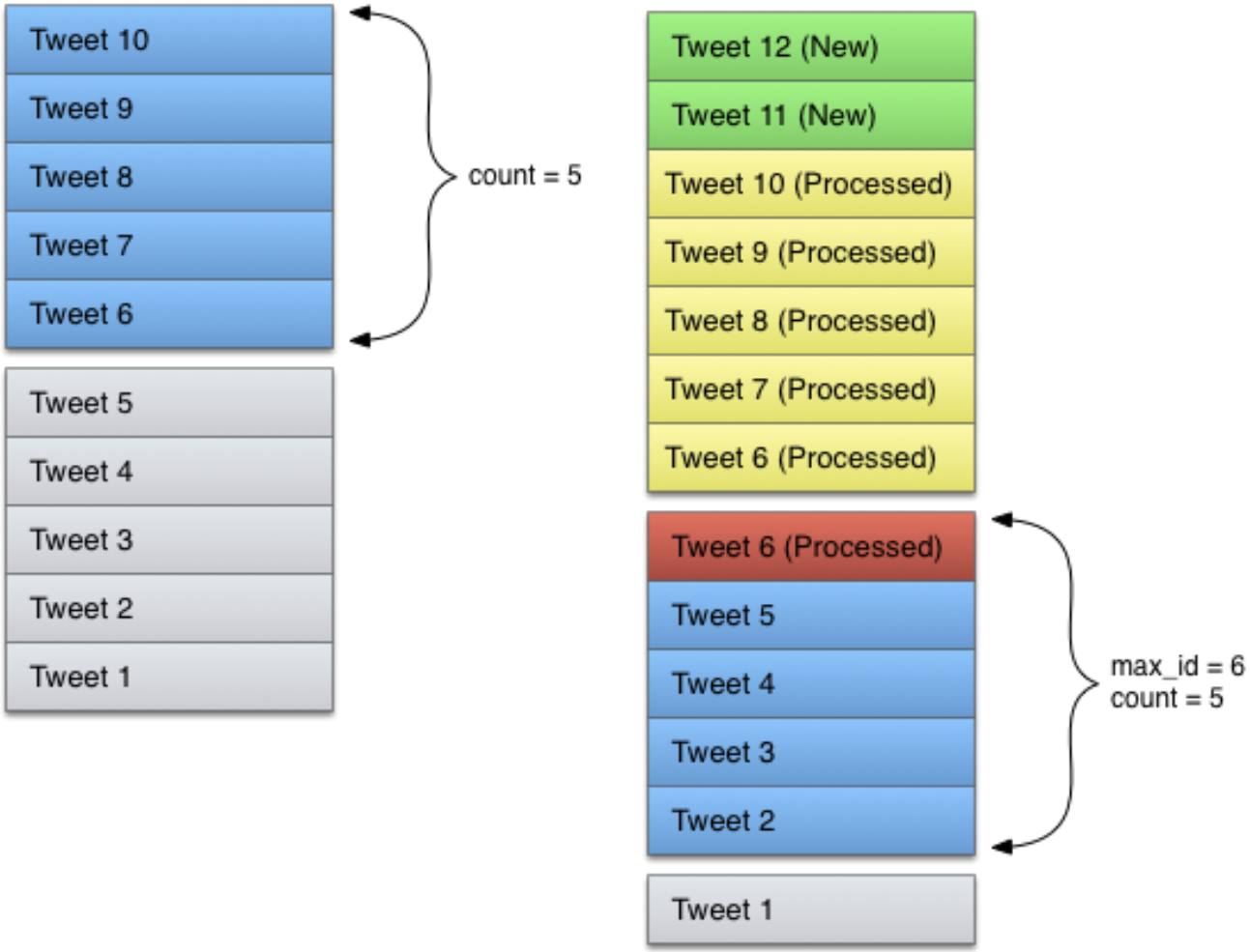
Retrieving Additional Results

- New Tweets added to the front



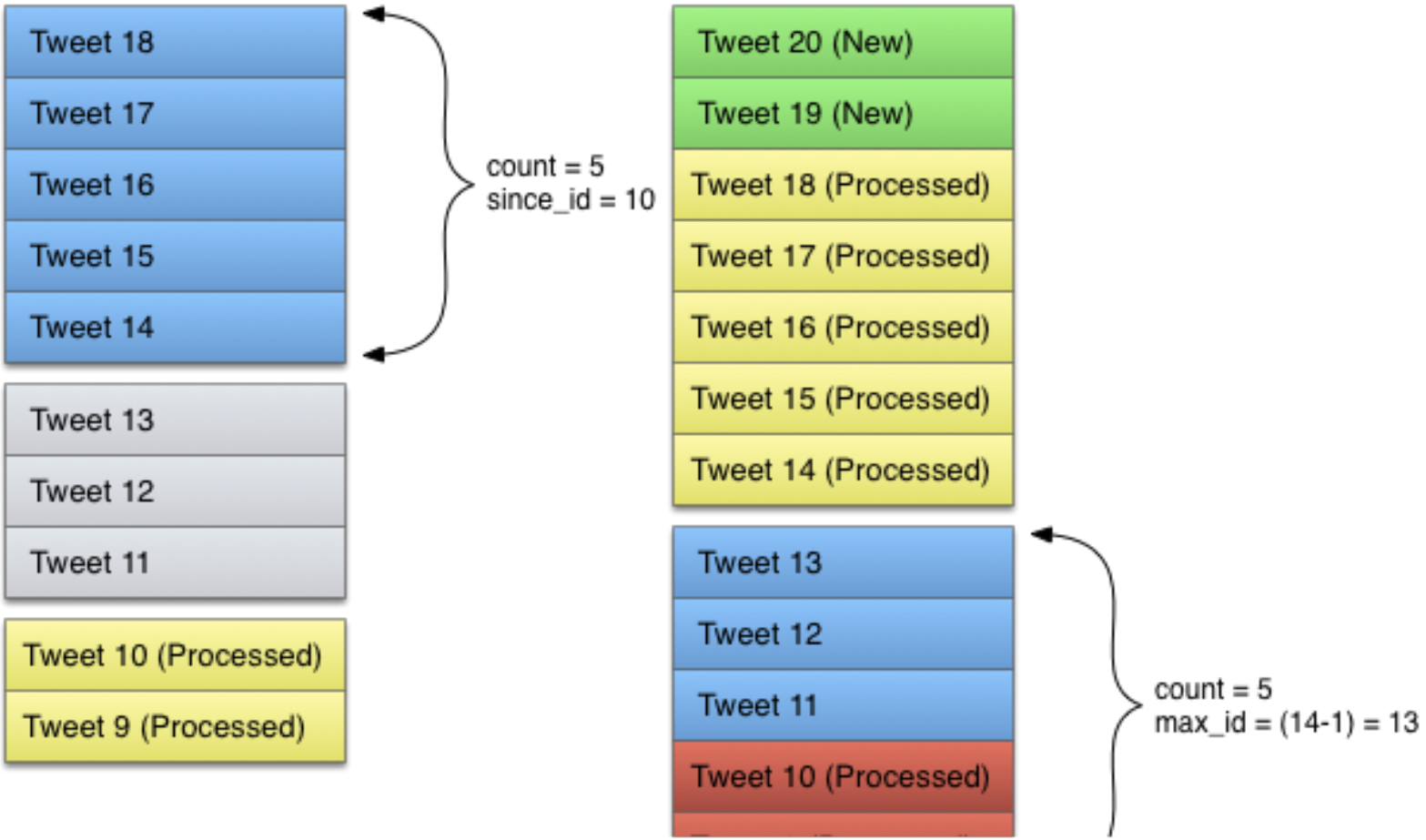
Retrieving Additional Results

- The `max_id` parameter can be helpful:



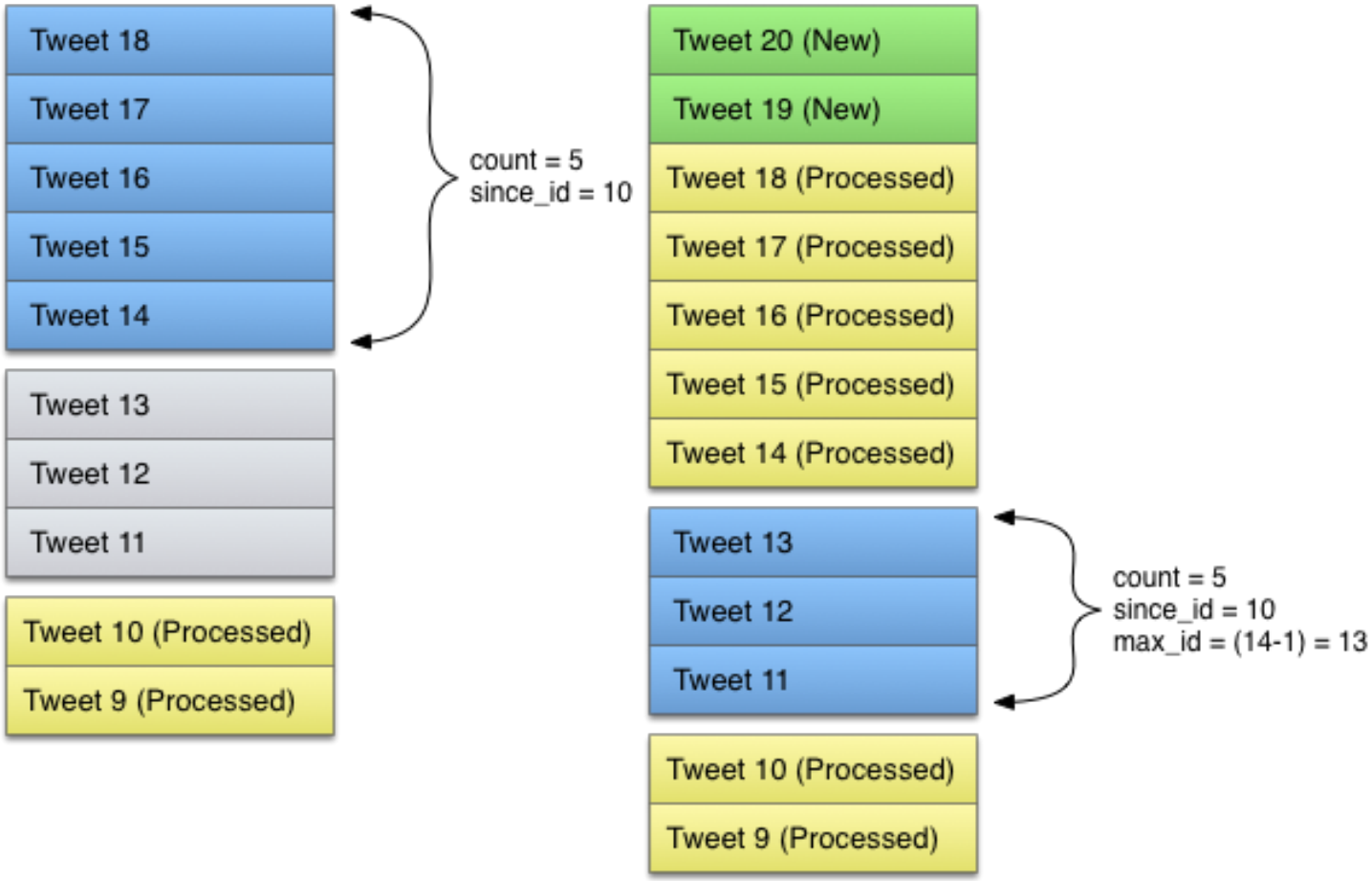
Retrieving Additional Results – Improved efficiency

- Use the `since_id` for the greatest efficiency



Retrieving Additional Results – Minimize redundand data

- Use both `max_id` and `since_id` to minimize amount of redundant data for the greatest efficiency.



Searching Tweets

```
for status in api.search("#RetakeInAction"):
    # process status here
    print(status.text)
    print(status.user.screen_name)
    print(status.created_at, "Fav: ", status.favorite_count)
    print("---")
```

Searching Tweets using Cursor

```
for status in tweepy.Cursor(api.search, q="#RetakeInAction").items():
    # process status here
    print(status.text)
    print(status.user.screen_name)
    print(status.created_at, "Fav: ", status.favorite_count)
    print("---")
```

Rate Limits

- Every day many thousands of developers make requests to the Twitter API.
- Limits are placed on the number of requests that can be made.
- The maximum number of requests that are allowed is based on a time interval, some specified period or window of time.
- The most common request limit interval is fifteen minutes.
- If an endpoint has a rate limit of 900 requests/15-minutes, then up to 900 requests over any 15-minute interval is allowed.
- Rate limits are applied based on which authentication method you are using.
  1. OAuth 2.0 Bearer Token: per-developer App
  2. OAuth 1.0a User Context: per-set of user access token. Users' rate limits are shared across all apps that they have authorized and the Twitter application.

Rate Limits – Response codes

- When an application exceeds the rate limit for a given Twitter API endpoint, the API will return a:
  HTTP 429 "Too Many Requests"
- When a "too many requests" or rate-limiting error occurs, the frequency of making requests needs to be slowed down.
- When a rate limit error is hit, the `x-rate-limit-reset`: HTTP header can be checked to learn when the rate-limiting will reset.

Twitter API v2 rate limits

Endpoint	Per app	Per user
Tweet lookup	300	900
Recent search	450	180
Timelines		
- User Tweet timeline	1500	900
- User mention timeline	450	180
Filtered stream		
- Connecting	50	
- Adding/deleting filters	450	
- Listing filters	450	
Sampled stream	50	
Hide replies		50
User lookup	300	900
Follows lookup	15	15

Amazon Simple Queue Service (SQS)

- Fully managed message queues for microservices, distributed systems, and serverless applications.
- Send, store, and receive messages between software components at any volume, without losing messages or requiring other services to be available.
- SQS offers two types of message queues.
  1. Standard queues offer maximum throughput, best-effort ordering, and at-least-once delivery.
  2. FIFO queues are designed to guarantee that messages are processed exactly once, in the exact order that they are sent.

AWS SQS Benefits

- Eliminate administrative overhead
- Reliably deliver messages
- Keep sensitive data secure
- Scale elastically and cost-effectively

Creating a new queue

```
# Get the service resource
sqs = boto3.resource('sqs')

# Create the queue. This returns an SQS.Queue instance
queue = sqs.create_queue(QueueName='ADMTest',
                         Attributes={'DelaySeconds': '5'})

# You can now access identifiers and attributes
print(queue.url)
print(queue.attributes.get('DelaySeconds'))
```

List & Search Queues

```python
# Get the service resource
sqs = boto3.resource('sqs')

# Print out each queue name, which is part of its ARN
for queue in sqs.queues.all():
    print(queue.url)


# Get the queue. This returns an SQS.Queue instance
queue = sqs.get_queue_by_name(QueueName='ADMTest')

# You can now access identifiers and attributes
print(queue.url)
print(queue.attributes.get('DelaySeconds'))
```

Send Messages

```
# Get the queue
queue = sqs.get_queue_by_name(QueueName='ADMTest')

# Create a new message
response = queue.send_message(MessageBody='world')

# The response is NOT a resource, but gives you a message ID and MD5
print(response.get('MessageId'))
print(response.get('MD5OfMessageBody'))
```

Send Messages with Custom Attributes

```
# Get the queue
queue = sqs.get_queue_by_name(QueueName='ADMTest')

# Create a new message
queue.send_message(MessageBody='boto3', MessageAttributes={
    'Author': {
        'StringValue': 'Daniel',
        'DataType': 'String'
    }
})

# The response is NOT a resource, but gives you a message ID and MD5
print(response.get('MessageId'))
print(response.get('MD5OfMessageBody'))
```

Send Messages in Batches

```
response = queue.send_messages(Entries=[
    {
        'Id': '1',
        'MessageBody': 'world'
    },
    {
        'Id': '2',
        'MessageBody': 'boto3',
        'MessageAttributes': {
            'Author': {
                'StringValue': 'Daniel',
                'DataType': 'String'
            }
        }
    }
])
# Print out any failures
print(response.get('Failed'))
```

Processing Messages

```
# Process messages by printing out body and optional author name
for message in queue.receive_messages(MessageAttributeNames=['Author']):
    # Get the custom author message attribute if it was set
    author_text = ''
    if message.message_attributes is not None:
        author_name = message.message_attributes.get('Author')
                                                .get('StringValue')
        if author_name:
            author_text = ' ({0})'.format(author_name)

    # Print out the body and author (if set)
    print('Hello, {0}!{1}'.format(message.body, author_text))

    # Let the queue know that the message is processed
    message.delete()
```