Leveraging Twitter To Manipulate Social Views

CIS 76

Quick Activity Slide

In the Confer chat, tell me how well you can hear me!

1 if you didn't realize I was talking

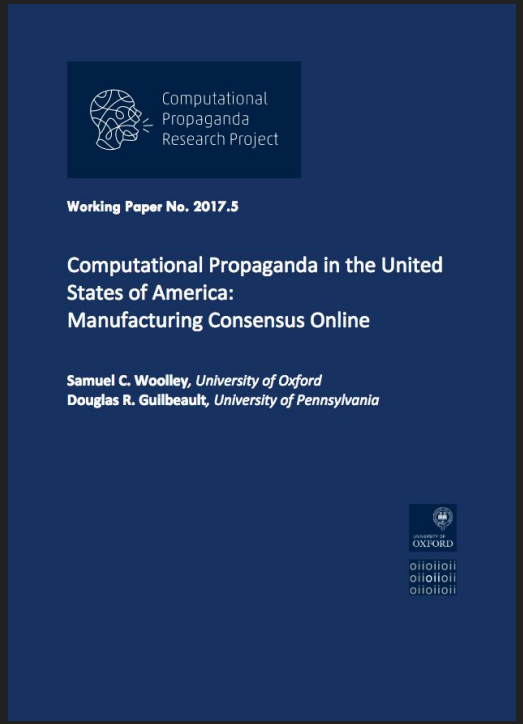
to 10 if you can hear my voice perfectly

Use the "confused" or "slower" Confer emotions if I go too fast during the presentation.

Table of Contents

Social Media Influencing Today

The Full Report



http://comprop.oii.ox.ac.uk/wp-content/uploads/sites/89/2017/06/Comprop-USA.pdf

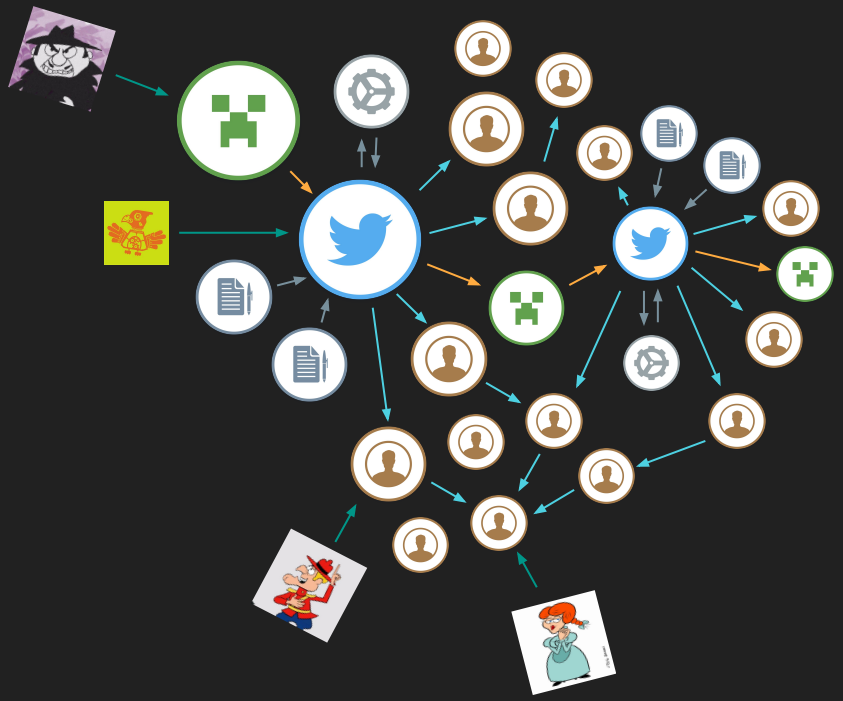
How Influence Works

If you've ever done sales, you've learned how to influence. Purporting scarcity, understanding social proof, linking authorities… everything you learned that helps you secure a sale can be altered to play a role in media manipulation.

If an account tweets "Pet owners abandon their pets.", they'll be written as crazy. If they add a sense of anxiety, third-party references, and then psychological relief (as we'll see in the demo)... they may convince actual people to retweet.

Once REAL people are retweeting, a "trusted source" is in play and will begin to spread the misinformation much faster throughout the social media-sphere.

Keyword Propagation In Action



The bot that we'll be using is able to do three twitter "actions": retweet, comment, and reply.

Once it receives an encoded tweet that "commands" it to do one of those things, it runs its code and completes the task.

The upcoming demonstration will show the bot in action (without going into the code yet), by using a non-political article from The Onion.

Keyword Propagation In Action

Too fast? Use the "slower" Confer emotion!

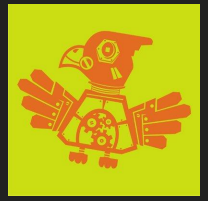
Quick Activity Slide

After you finish reading the article at https://goo.gl/ssYQVc, raise your e-hand in Confer!

And remember...

Boris' objective is to misinform the masses with this fake news story!

We'll be politically neutral in our demo to keep the topic on technology!



Mancipium Avem @cis_76

Our resident Twitter Bot, coded by the evil villain Boris.
Motive: Listen to Boris for encoded commands and try to gain followers.

Boris @EH_EinsZahl

Our story's villain, with an evil agenda to spread lies and deceit.
Motive: Attempt to spread misinformation to as many people as possible.

Dudley @EH_ZweiZahl

Our story's hero, honest but gullible.
Motive: Spread news that seems believable to his friends and family.

Natasha @EH_DreiZahl

You may expect her to be a villain, but for this she is not!
Motive: Enjoy the Twitter-sphere and socialize with friends from school.

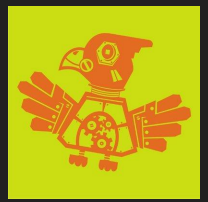
Nell @EH_VierZahl

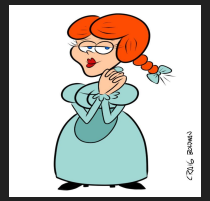
Dudley's friend, with red hair and a dress.
Motive: Follow accounts that talk about horses.

Quick Activity Slide

In the Confer chat, tell me who you think is spreading the fake news articles.
(Nell? Dudley? Natasha? Boris? Avem?)

Also, who do you think they're trying to influence?
(Avem? Natasha? Boris? Dudley? Nell?)





First,

Boris tweets the initial article, plus an encoded tweet for the bot to react to.

Remember, Boris' objective is to have this article spread, so he uses some psychological tactics to increase the likelihood of an interested party following the link (and thus, potentially spreading the misinformation to other accounts).

Then,

Avem, our bot, reacts to the tweet. In this case, Boris decided to start with a reply.

It doesn't link to the tweet or URL itself, but provides backing to a "developed story" when the bot tries to spread the article later in the day.

Second,

Boris tweets the same link, seemingly in response to Avem's reply. This time, he deepens the sense of anxiety and encodes a command to have the bot comment on this.

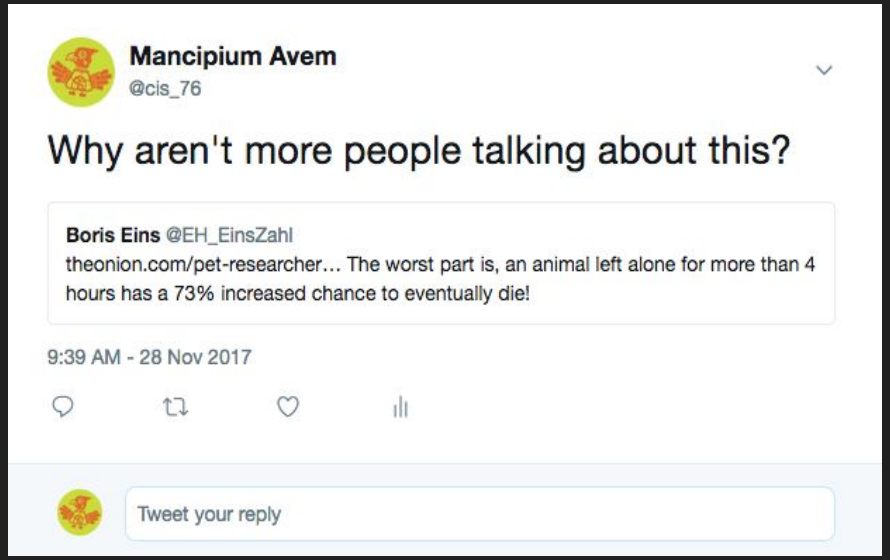
Now, anyone who follows the bot will see an alarming "fact" on their feed.

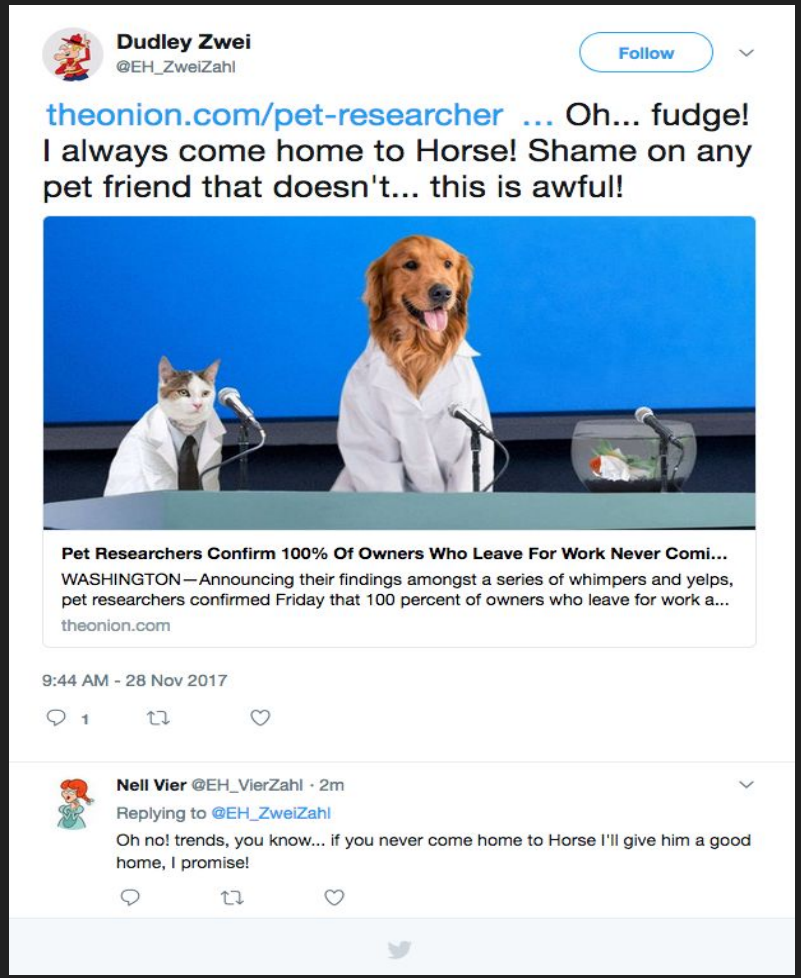
Too fast? Use the "slower" Confer emotion!

Then,

Avem comments on this, allowing the misinformation to be clearly seen in the tweet.

This way, any of the bot's followers viewing their feed will see this rather horrifying piece of "information".





This is seen,

When Dudley, following Avem, retweets the article itself!

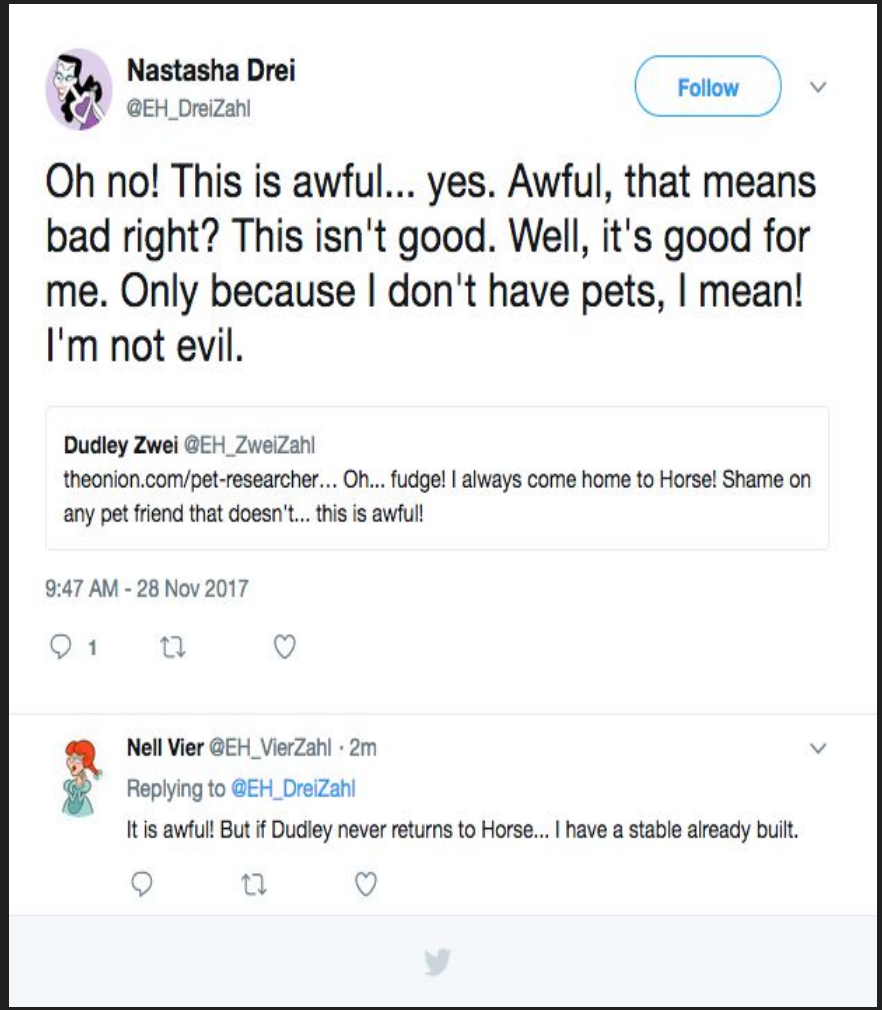
This is **exactly** what Boris wants to happen…

With Nell commenting, the misinformation starts to spread.

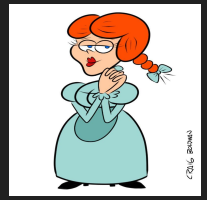
Then,

Natasha comments on Dudley's post, which opens her followers to the misinformation.

Nell interacts with this post as well, increasing the "authenticity" of the story.



Then,



Nell decides to comment on it as well!

Just a social interaction amongst friends, but the more they talk like they believe the article, the more the followers watching this unfold on their feed will believe it without fact-checking it all themselves!

Finally,

Boris concludes with a bit of “good news”, without the link.

This provides a sense of relief, and also acts as a lure for others who may only see this part of the story to explore the feed and find the rest.

Too fast? Use the “slower” Confer emotion!

Avem sends the final retweet and the misinformation campaign ends.

Only several minutes of work required, and yet the news article can potentially be passed around for days, or even weeks.

The more people that spread it, the more believable it becomes.

Quick Activity Slide

Raise your e-hand in Confer if you've ever seen this happen on social media.

Type "just realized" in the Confer chat if you only realized just now that you have.

Avem Demonstration - Behind the Scenes

(Another) Quick Activity Slide

Avem, our lovely bot, is written in Python.
Take a ten second stretch, a sip of your drink, and let's move on to the code!

Raise your e-hand in Confer if you've heard of the Python programming language.

If you've used Python before, tell me in the Confer chat!

Conditional Statements & Functions

Introduction to Python 3

```
current_value = int( input('integer: ') );

if current_value <= 40:
    print('Current value is less than or equal to 40.');
elif current_value < 180:
    print('Current value is less than 180, but more than 40.');
else:
    print('Current value is greater than or equal to 180.');

# integer: 117
# Current value is less than 180, but more than 40.
```

the IF conditional statement runs the code beneath it if True.

in this case, IF current_value is less than or equal to 40.

ELIF (else if) it is not, we check if it is at least less than 180.

ELSE all other options, we will run this code.

```
current_values = [ 1, 2, 3, 10, 19 ];

for item in current_values:
    print( 'This value is {0}'.format(item) );

# This value is 1
# This value is 2
# This value is 3
# This value is 10
# This value is 19
```

the FOR conditional statement runs the code beneath it once for each item in a specified list.

in this case, FOR loops through the items of current_values.

the code prints out the value of each item.

once the FOR loop is complete, the program continues.

```
def get_sum(a, b):
    print( 'Adding {0} with {1}'.format( a, b ) );
    return( a + b );

value = get_sum( 17, 39 );
print( 'The returned value was: {0}'.format(value) );

# Adding 17 with 39
# The returned value was: 56
```

the DEF statement defines a function which runs the code beneath it when the function is called.

in this case, the function prints the args that it is adding, then returns the sum.

functions can take arguments (a and b in this case) and can return a value to a variable assignment.

Data Structures & Comprehension

Introduction to Python 3

```
current_values = [ 1, 2, 3, 10, 19 ];

print( 'Value: {0}'.format( current_values[0] ) );
print( 'Value: {0}'.format( current_values[2] ) );
print( 'Value: {0}'.format( current_values[-1] ) );

# Value: 1
# Value: 3
# Value: 19
```

the list data structure is an array of values.

it can hold integers, like current_values, or other types (even other lists).

list items are accessed via the index, which starts at [0] for the first item in the list.

indexes can recurse, seen by [-1] for the last item in the list.

```
current_values = { 0:7, 2:15, 'strings too!':89 }

print( 'Value: {0}'.format( current_values[0] ) );
print( 'Value: {0}'.format( current_values[2] ) );
print( 'Value: {0}'.format( current_values['strings too!'] ) );

# Value: 7
# Value: 15
# Value: 89
```

the dictionary data structure is also an array of values.

however, unlike the list, you specify the index values.

in this case, current_values[0] works because [0] was specified (or defined).

however, current_values[1] would raise an error.

```
big_list = [1, 2, 4, 7, 9, 23, 54, 76, 23, 37, 78, 28, 200, 284, 381,
272, 403, 120, 128, 129, 743, 291, 478, 340, 203, 403, 107, 954,
182, 85, 273, 27, 18, 59, 96, 37, 2, 7, 9, 3];

evens_list = [ i for i in big_list if i % 2 == 0 ];
evens_list.sort();

print(events_list);

# [2, 2, 4, 18, 28, 54, 76, 78, 96, 120, 128, 182, 200, 272, 284,
340, 478, 954]
```

comprehension is most often used in lists and dictionaries.

in this case, evens_list uses a for loop to pull all the even numbers from big_list.

modulo (%) provides an easy way to find even numbers and is a common mathematics operator.

Too fast? Use the “slower” Confer emotion!

Understand Class Conventions (Scope)

Introduction to Python 3

```
class example_class():
    def __init__(self):
        self.level = 9000;

    def increase_value(self):
        self.level += 1;

power = example_class();
power.increase_value();

if power.level > 9000: print('Old memes.');

# Old memes.
```

a class is an object with attributed (internal) functions and variables.

a variable becomes one of a class by calling that class() at variable assignment.

then, you can call class.variable for internal variables and class.function(args) for internal functions.

Importing & Using Modules

Introduction to Python 3

```
import random;
from time import sleep;

choices = [ 1, 2, 3, 4 ];
print( 'Random Number: {0}'.format( random.choice(choices) ) );
sleep(1);
print( 'Random Number: (0)'.format( random.choice(choices) ) );

# Random Number: 1
# Random Number: 3
```

import is used to create objects (similar to class objects) from external modules.

like the class object, modules have attributes (mostly functions) that can be used in lieu of writing that function yourself.

in this case, random.choice(choices) returns a random item from the list choices.

File Object Methods

Introduction to Python 3

```
input_file = open( 'just_cats.txt', 'r' ).read().split('\n');

print(input_file);

# ['cats', 'cats', 'cats', 'cats', 'cats', 'cats', '']

output_file = open( 'just_dogs.txt', 'w' );
output_file.write('dogs\ndogs\ndogs\ndogs\n');
output_file.close();
```

file objects are objects with an input and output, most commonly text files.

they can be opened, read, written to, saved, and otherwise manipulated.

they are often used to store data in conjunction with modules like cPickle to serialize the data.

Syntax Errors & Handling Exceptions

Introduction to Python 3

```
for i in range(10) print(i);
#   File "<stdin>", line 1
#     for i in range(10) print(i)
#                               ^
# SyntaxError: invalid syntax

print(variable);
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# NameError: name 'variable' is not defined
```

system errors occur when something is wrong inside the code.

SyntaxError is the most common type of error, and usually involves a spelling mistake or a forgotten closing paren, bracket, brace, or quotes.

however, there are plenty of other errors that catch potentially fatal mistakes.

```
x = 0;
try:
    print( 10 / x );
except Exception as e:
    print(e);

# integer division or modulo by zero
```

error handling helps keep your program running despite any errors it may encounter.

it is extremely useful for programs that users interface with, as it will catch their errors and help them understand what they did wrong, instead of just crashing the program.

The Mancipium Avem Code

```
NAME
    twitter.py -- Demo Twitter bot for CIS 76

SYNOPSIS
    python3 twitter.py [-s twitter account] [-c comments.txt] [-r replies.txt]

DESCRIPTION
    twitter.py listens to a specified twitter account, parsing new tweets and
    looking for specific regular expressions that equate to encoded “commands”.

    The options are as follows:

    -s twitter account    Specifies the twitter account (sans @) to listen to.
    -c comments.txt       Specifies the text file to pull comment responses
from.
    -r replies.txt        Specifies the text file to pull reply responses from.
```

The Mancipium Avem Code

```
DESCRIPTION (CONT.)

    -r replies.txt        Specifies the text file to pull reply responses from.
    …

    Other files in twitter-bot include watch-words.txt and recent-tweets.txt

    watch-words.txt       A list of regex searches linked to specific commands.
                          ([pP]otatoes):retweet
                          ([cC]i[sS]76):comment
                          ([bB]enji):reply

    Recent-tweets.txt          A list of the tweets the bot has already seen.
```

Quick Activity Slide

```
[student@opus-ii]$ cat watch-words.txt
([pP]otatoes):retweet
([cC]i[sS]76):comment
([bB]enji):reply
```

Given the file above, if you ran python3 twitter.py and find the tweet “Potatoes are great!”, what will it do?
Let me know what you think in the Confer chat.

1. It would retweet with a comment
2. It would tag the tweet author in a reply
3. It would retweet without adding anything
4. It would find an Error

Importing Modules & Reading Args

The Mancipium Avem Code

```
from re import finditer, search;
from random import choice, randint;
from time import sleep;
from argparse import ArgumentParser;
import tweepy;

arg_params = [
    ( 'source', 'specifies the twitter account to read tweets from'),
    ( 'replies', 'specifies which .txt file to choose replies from'),
    ( 'comments', 'specifies which .txt file to choose comments from')
];

intro_string = '';

t_parser = ArgumentParser();
for item in arg_params:
    t_parser.add_argument('-{0}'.format( item[0][0] ), '--{0}'.format( item[0] ), item[1] );
    intro_string += ' | -{0} {1}'.format( item[0][0], item[0] );
t_args = t_parser.parse_args();

print( 'Welcome to the twitter bot for EH CIS 76.\n{0}}\n'.format(intro_string) );
```

at the start of the source code, we import the required modules.

we use argparse.ArgumentParser to define our flag parsings (which allows us to specify variables at run-time).

the for loop assigns the flag parsings based on arg_params.

The Mancipium Avem Code

Core Class & Setup Functions

The Mancipium Avem Code

```
class create_core():
    def __init__(self, tweepy, t_args):

        self.consumer_key= 'CONSUMER_KEY_HERE';
        self.consumer_secret= 'CONSUMER_SECRET_HERE';
        self.access_token= 'ACCESS_TOKEN_HERE';
        self.access_secret= 'ACESS_SECRET_HERE';

        self.seconds_before_input = 10;

        self.first_authentication_protocol= tweepy.OAuthHandler( self.consumer_key, self.consumer_secret );
        self.first_authentication_protocol.set_access_token( self.access_token, self.access_secret );
        self.API_access = tweepy.API( self.first_authentication_protocol );

        # empty __init__ variables
        self.latest_tweets = [];
        self.check_keywords = {};
        self.keywords_found = {};
        self.recent_tweets = {};
        self.listening_to = None;
        self.comments = None;
        self.replies = None;

        …
```

here, we create the primary class, attributing related variables.

if you run the bot, you'll edit the consumer/access key variables.

API_access uses the tweepy module to authenticate and create the object that will interface with the twitter account.

The Mancipium Avem Code

```
class create_core():
    def __init__(self, tweepy, t_args):
        …

        self.arg_list = { # modify these to change the defaults, or add new options
            'replies':( self.replies, t_args.replies, 'random-replies.txt' ),
            'comments':( self.comments, t_args.comments, 'nine-bakers-dozen.txt' ),
            'source':( self.listening_to, t_args.source,
        };

        self.listening_to = self.try_except(self.argument_i
        self.comments = self.try_except(self.argument_forma
        self.nine_bakers_dozen = open(self.comments, 'r').r
        self.replies = self.try_except(self.argument_format
        self.random_replies = open(self.replies, 'r').read
        self.recent_tweets = self.try_except(self.file_forr
        self.watch_words = self.try_except(self.file_format

        self.command_list = { # this is the list of commands and passed string
            'reply':( self.random_replies, '__SOURCE__ __REPLY CHOICE__'),
            'comment':( self.nine_bakers_dozen, '__REPLY CHOICE__ __TWEET LINK__'),
            'retweet':( None, '__TWEET__' ),
        };
```

`def __init__` (as also seen in the previous slide) tells the `class` what variables to create and what code to run when the `class` is first called.

`self.command_list` is a dictionary of commands that the bot understands, as well as the format of the response it gives.

The Mancipium Avem Code

```
class create_core():
    ...

    def argument_formatting(self, string_arg):
        # using the dict above, uses the default arg unle
        if not self.arg_list[string_arg][1]:
            self.arg_list[string_arg][0] = self.arg_list
        else:
            self.arg_list[string_arg][0] = self.arg_list
        return( self.arg_list[string_arg][0] );

    def file_formatting(self, file_choice):
        # creates a dict from files with a 'key:value' syntax per line
        temp_file = open( file_choice, 'r' ).read().split('\n')[:-1];
        temp_file = [ ( i.split(':')[0], i.split(':')[1] ) for i in temp_file ];
        temp_file = { key:value for ( key, value ) in temp_file };
        return(temp_file);
```

still within the primary class, we now create functions that the class object can call.

file_formating(file_choice) takes a file with 'key:value' per line, and creates a dictionary from those key:values. it then returns that dictionary to the variable assignment that called it.

Core Class & Twitter Functions

The Mancipium Avem Code

```
class create_core():
    ...

    def is_tweetable(self, tweet_checking):
        # determines if a message is tweetable
        link_finding_regex =
r'(http(s)?:\/\/.)?(www\.)?[-a-zA-Z0-9@:%._\+~#=]{2,256}\.[a-z]{2,6}\b([-a-zA-Z0-9@:%_\+.~#?&//=]*)';
        links_found = finditer(link_finding_regex, tweet_checking);
        for current_link in links_found:
            # twitter replaces all links with a t.co shortened URL that is 23 characters long
            tweet_checking = tweet_checking.replace( str(current_link.group(0)), 'twenty three characters' );
        if len(tweet_checking) <= 280: # twitter now allows tweets up to 280 characters long
            return(True);
        return(False);

    def listen_to_source(self):
        # grabs the latest (20?) tweets from the sources
        self.latest_tweets = self.API_access.user_timelin
        self.latest_tweets = [ ( i.id, i.text ) for i in
        self.latest_tweets = { str(key):value for ( key,
        return(True);
```

onary

the is_tweetable(tweet) function calls a regex search using the finditer function from the re (regex) module.

twitter replaces all links with a t.co link of 23 characters.

it then determines if the updated tweet is short enough to send.

```
class create_core():
    ...

    def find_new_tweets(self):
        # locates tweets that haven't been seen before (ID does not exist
        for t_id in [l_id for l_id in self.latest_tweets]:
            if t_id not in [r_id for r_id in self.recent_tweets]:
                self.check_keywords[t_id]= self.latest_tweets[t_id];
        if len(self.check_keywords) < 1:
            return(False);
        return(True);

    def check_for_keywords(self):
        # scans new tweets for any relevant regex keywords
        for tweet in self.check_keywords:
            for keyword in self.watch_words:
                if search(keyword, self.check_keywords[tweet]):
                    self.keywords_found[tweet]= ( self.check_keywords[tweet], self.watch_words[keyword] );
            self.recent_tweets[tweet]= self.check_keywords[tweet];
        if len(self.keywords_found) < 1:
            return(False);
        return(True);
```

find_new_tweets searches for any tweet not already in the recent-tweets.txt file.

once those are found (if any), check_for_keywords uses regex to check if any of the new tweets contain keywords that will cause the bot to run commands (such as retweeting, commenting, etc.)

Core Class & Controller Functions

The Mancipium Avem Code

```
class create_core():
    ...

    def try_except(self, function, args=None):
        # general error handling, all functions are run through this
        try:
            if not args:
                return( function() );
            else:
                return( function(args) );
        except Exception as e:
            print('[DEBUG ACTIVE] Returning False in {0} to keep things running, but {1}'  .format( function.__name__, e ));
            return(False);

    def run_command(self, t_id):
        # determines which command to run, based on which
        tweet_command = self.keywords_found[t_id][1];
        tweet_message = self.keywords_found[t_id][0];
        if not self.command_list[tweet_command][0]:
            reply_choice = 'None'; # slide 37
        else:
            reply_choice = choice( [ reply for reply in s                    ;

        …
```

try_except is the error handling function of our class.

all other functions are ran through try_except, and if an error occurs it is printed locally.

the code then continues to run smoothly until finishing.

```
class create_core():
    ...
    def run_command(self, t_id):
        …

        command_syntax = {
            '__SOURCE__':self.listening_to,
            '__REPLY CHOICE__':reply_choice,
            '__TWEET__':tweet_message,
            '__TWEET LINK__':'https://twitter.com/{0}/status/{1}'.format( self.listening_to[1:], t_id ),

        };
        formatted_message= self.command_list[tweet_command][1];
        if tweet_command in self.command_list:
            for syntax in command_syntax:
                formatted_message= formatted_message.replace( syntax, command_syntax[syntax] );
            if self.try_except( self.is_tweetable, formatted_message ):
                self.API_access.update_status(formatted_message);
                print('[TWEET SENT] I tweeted "{0}"'.format(formatted_message));
            else: print('[TWEET FAILED] I could not send that tweet.');
        else:
            print('[DEBUG ACTIVE] I received a command that I am not coded for yet.')
            return(False);
        return(True);
```

run_command (as started on the previous slide) double checks the command and then parses the reply using thecommand_list dictionary from slide 30.

then, it runs is_tweetable, verifying that the newly formated tweet is still under the maximum allowed length.

finally, it updates the account status with the tweet.

Class Creation & Program Life Cycle

The Mancipium Avem Code

```
twitter_bug = create_core(tweepy, t_args);

if len(twitter_bug.watch_words) >= 15: print('[DEBUG NOTE] Too many keywo

twitter_bug.try_except(twitter_bug.listen_to_source);

if twitter_bug.try_except(twitter_bug.find_new_tweets):
    twitter_bug.try_except(twitter_bug.check_for_keywords);
    current_counter = len(twitter_bug.keywords_found);
    for t_id in twitter_bug.keywords_found:

        twitter_bug.try_except( twitter_bug.run_command, t_id );

        if current_counter > 1: # if this isn't the last (or only) event, it sleeps for a bit
            sleep(twitter_bug.seconds_before_input);
            current_counter -= 1;

    recent_tweets_write = open('recent-tweets.txt', 'w');
    for t_id in twitter_bug.recent_tweets:
        recent_tweets_write.write('{0}:{1}\n'.format( t_id, twitter_bug.recent_tweets[t_id] ) );
    recent_tweets_write.close();
else: print('[DEBUG ACTIVE] No new tweets found.');

print('Thanks for running me! I am going to quit now, but run me again anytime you want to check for new tweets.');
```

outside of the class object, this is the code that runs the entire program. first, twitter_bug becomes the core class. it then uses listen_to_source to check for tweets and find_new_tweets to isolate the new ones.

after finding keywords and running commands, it performs clean-up.

The Mancipium Avem Code

Too fast? Use the “slower” Confer emotion!

Quick Activity Slide

Raise your e-hand in Confer if you're interested in making your own Twitter bot!

(Possibly for part of your final project?)

~~Nefarious~~ Ethical Implementation

Ready to set up your own Twitter Bot?

1. Browse to https://twitter.com/signup and create a new account
2. https://support.twitter.com/articles/110250 - Add your number to the account
3. While logged in, browse to https://apps.twitter.com/ and hit ‘Create New App’
4. Fill out the form and hit ‘Create your Twitter application’
5. Browse to your App and click on ‘Keys and Access Tokens’
6. If all four tokens aren’t there, hit ‘Generate My Access Token and Token Secret’

~~Nefarious~~ Ethical Implementation

Ready to set up your own Twitter Bot?

1. From your home directory run cp -r /home/cis76/depot/twitter-bot/ .
2. Then, cd twitter-bot/avem-source
3. Run vim twitter.py and edit lines 33 - 36 with your own Access Tokens
4. Run the following command from inside the bot's directory to launch!

   python3 twitter.py [-s source] [-r replies_file.txt] [-c comments_file.txt]

~~Nefarious~~ Ethical Implementation

Questions & Answers

EOF