SENDING EMAIL AND TEXT MESSAGES

CS 3030: Python

Instructor: Damià Fuentes Escoté

UCCS University of Colorado Colorado Springs

Sending email and text messages

- Checking and replying to email is a huge time sink.
- Of course, you can’t just write a program to handle all your email for you, since each message requires its own response.
- But you can still automate plenty of email-related tasks once you know how to write programs that can send and receive email.

Sending email and text messages - Examples

- Example 1
  - *You have a spreadsheet full of customer records and want to send each customer a different form letter depending on their age and location details. You can write your own program to send these emails, saving yourself a lot of time copying and pasting form emails.*
- Example 2
  - *Write programs to send emails and SMS texts to notify you of things even while you're away from your computer. If you're automating a task that takes a couple of hours to do, you don't want to go back to your computer every few minutes to check on the program's status. Instead, the program can just text your phone when it's done—freeing you to focus on more important things while you're away from your computer.*

SMTP

- Much like HTTP is the protocol used by computers to send web pages across the Internet, **Simple Mail Transfer Protocol (SMTP)** is the protocol used for sending email.
- SMTP dictates how email messages should be formatted, encrypted, and relayed between mail servers, and all the other details that your computer handles after you click Send.
  - *You don't need to know these technical details, though, because Python's smtplib module simplifies them into a few functions.*
- SMTP just deals with sending emails to others. A different protocol, called IMAP, deals with retrieving emails sent to you.

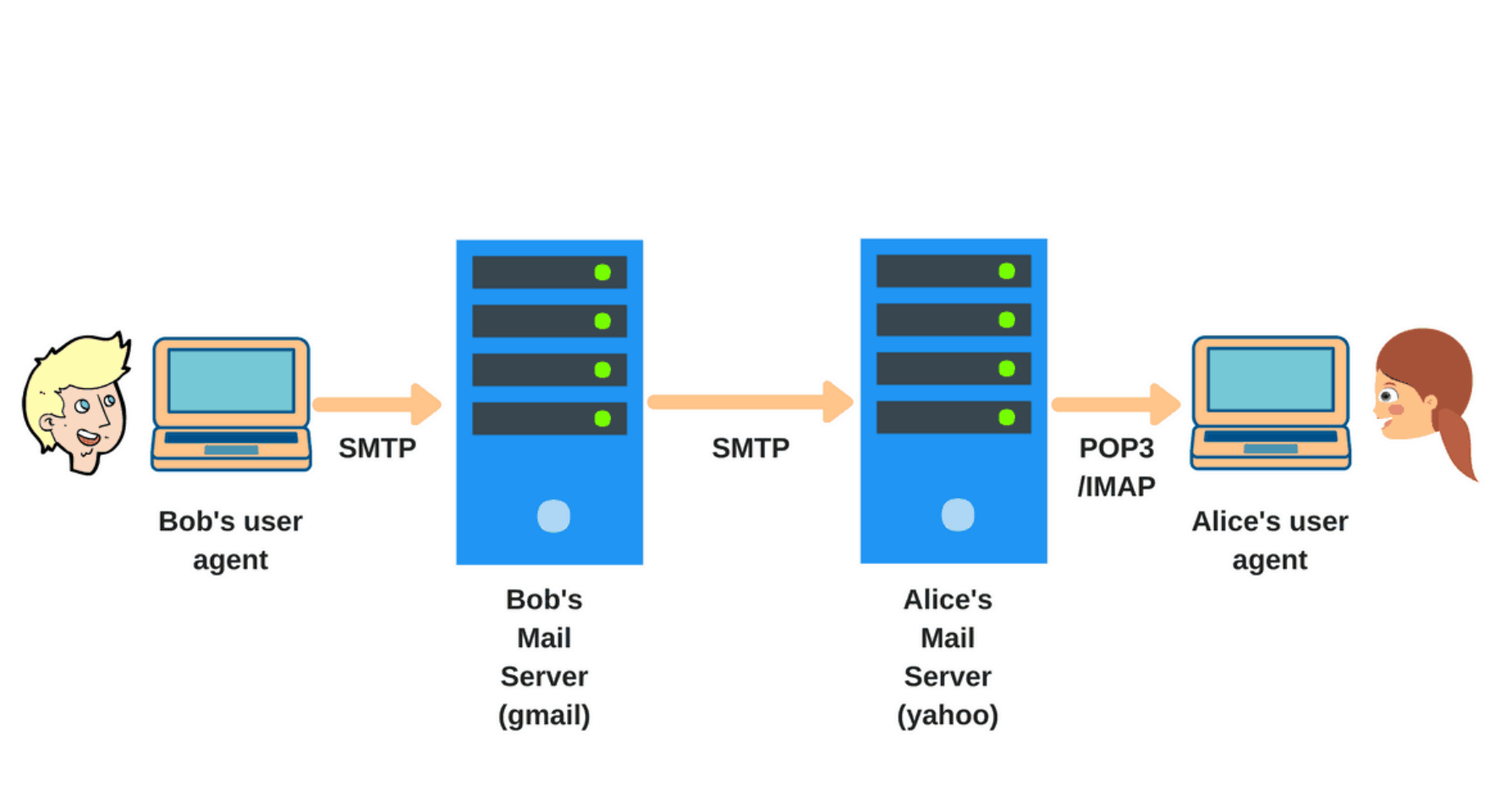
IMAP

- Just as SMTP is the protocol for sending email, the Internet Message Access Protocol (IMAP) specifies how to communicate with an email provider's server to retrieve emails sent to your email address.
  - *pip install imapclient*
  - *http://imapclient.readthedocs.org/ .*
- The imapclient module downloads emails from an IMAP server in a rather complicated format. Most likely, you'll want to convert them from this format into simple string values.
- The pyzmail module does the hard job of parsing these email messages for you.
  - *pip install pyzmail36*
  - *import pyzmail*
  - *http://www.magiksys.net/pyzmail/ .*

SMTP and IMAP - Concepts

- Let's say Bob wants to send and email to Alice
- Bob's **user agent**
  - *This is the application running on Bob's laptop that he uses to compose, reply to, and read his email messages.*
    - Mail app, Python script, Outlook, Gmail, etc
- Bob's **mail server**
  - *Say Bob has an email account on gmail. What this means is that there is a remote machine under the gmail.com domain that manages all the email messages sent to Bob and all email messages sent from Bob to other users on other mail servers. This remote machine is what we call Bob's mail server*
- Alice's mail server
- Alice's user agent

SMTP and IMAP



SMTP

Sending Email

```
import smtplib

smtpObj = smtplib.SMTP('smtp.gmail.com', 587)
print(smtpObj.ehlo())
# (250, b'CY4PR02CA0026.outlook.office365.com Hello [2601:281:c70......
print(smtpObj.starttls())
# (220, b'2.0.0 SMTP server ready')
print(smtpObj.login('bob@gmail.edu', PASSWORD))
# (235, b'2.7.0 Authentication successful')
message = "Subject: Welcome to the Python course!!\n\nThis is a test message!"
print(smtpObj.sendmail('bob@gmail.edu', 'alice@yahoo.com', message))
# {}
print(smtpObj.quit())
# (221, b'2.0.0 Service closing transmission channel')
```

Sending Email – Connecting to an SMTP Server

```
smtpObj = smtplib.SMTP('smtp.gmail.com', 587)
```

- Configuring the SMTP server and port.
  - *These settings will be different for each email provider, but a web search for <your provider> smtp settings should turn up the server and port to use.*
  - *The port is an integer value and will almost always be 587, which is used by the command encryption standard, TLS.*
  - *If the smptlib.SMTP() call is not successful, your SMTP server might not support TLS on port 587. In this case, you will need to create an SMTP object using smtplib.SMTP_SSL() and port 465 instead.*

Table 16-1: Email Providers and Their SMTP Servers

Provider	SMTP server domain name
Gmail	*smtp.gmail.com*
Outlook.com/Hotmail.com	*smtp-mail.outlook.com*
Yahoo Mail	*smtp.mail.yahoo.com*
AT&T	*smpt.mail.att.net* (port 465)
Comcast	*smtp.comcast.net*
Verizon	*smtp.verizon.net* (port 465)

Sending Email – Sending the SMTP “Hello” Message

```
print(smtpObj.ehlo())
# (250, b'CY4PR02CA0026.outlook.office365.com Hello [2601:28.....
```

- The ehlo() method is used to “say hello” to the SMTP email server. This greeting is the first step in SMTP and is important for establishing a connection to the server.
- If the first item in the returned tuple is the integer 250 (the code for “success” in SMTP), then the greeting succeeded

Sending Email – Starting TLS Encryption

```
print(smtpObj.starttls())
# (220, b'2.0.0 SMTP server ready')
```

- If you are connecting to port 587 on the SMTP server (that is, you're using TLS encryption), you'll need to call the starttls() method next. This required step enables encryption for your connection. The 220 in the return value tells you that the server is ready.
- **If you are connecting to port 465 (using SSL), then encryption is already set up, and you should skip this step.**

Sending Email – Logging in to the SMTP Server

```
print(smtpObj.login('bob@gmail.edu', PASSWORD))
# (235, b'2.7.0 Authentication successful')
```

- Use the login() method by passing a string of your email address as the first argument and a string of your password as the second argument. The 235 in the return value means authentication was successful. Python will raise an smtplib .SMTPAuthenticationError exception for incorrect passwords.
- **Be careful about putting passwords in your source code**. If anyone ever copies your program, they'll have access to your email account! Solutions:
  - *Use input() and have the user type in the password.*
  - *Encrypt the password*

Sending Email – Sending an Email

```
message = "Subject: Welcome to the Python course!!\n\nThis is a test message!”
print(smtpObj.sendmail(‘bob@gmail.edu’, ‘alice@yahoo.com’, message))
# {}
```

- Call the sendmail() method to actually send the email. Three arguments:
  - *Your email address as a string (for the email’s “from” address)*
  - *The recipient’s email address as a string or a list of strings for multiple recipients (for the “to” address)*
  - *The email body as a string*
- The start of the email body string must begin with 'Subject: \n' for the subject line of the email. The '\n' newline character separates the subject line from the main body of the email.
- The return value from sendmail() is a dictionary. There will be one key-value pair in the dictionary for each recipient for whom email delivery failed . An empty dictionary means all recipients were successfully sent the email.

Sending Email – Disconnecting from the SMTP Server

```
print(smtpObj.quit())
# (221, b'2.0.0 Service closing transmission channel')
```

- Be sure to call the quit() method when you are done sending emails. This will disconnect your program from the SMTP server. The 221 in the return value means the session is ending.

Sending Email

```
import smtplib

smtpObj = smtplib.SMTP('smtp.gmail.com', 587)
print(smtpObj.ehlo())
# (250, b'CY4PR02CA0026.outlook.office365.com Hello [2601:281:c70......
print(smtpObj.starttls())
# (220, b'2.0.0 SMTP server ready')
print(smtpObj.login('bob@gmail.edu', PASSWORD))
# (235, b'2.7.0 Authentication successful')
message = "Subject: Welcome to the Python course!!\n\nThis is a test message!"
print(smtpObj.sendmail('bob@gmail.edu', 'alice@yahoo.com', message))
# {}
print(smtpObj.quit())
# (221, b'2.0.0 Service closing transmission channel')
```

IMAP

IMAPClient module issue

- In my computer I had this issue using Python 3.6 and 3.7
  - *ssl.SSLError: [SSL: CERTIFICATE_VERIFY_FAILED] certificate verify failed (_ssl.c:833)*

- It probably work in your computers as this depends on SSL certificate installation. If you go through the same issue, you could try with Python 2.7 as that version do not check the SSL certificate.

IMAP – Connecting to an IMAP Server

```
from imapclient import IMAPClient

imapObj = IMAPClient('imap-mail.outlook.com', ssl=True)
```

- IMAP server's domain name is different from the SMTP server's domain name

Table 16-2: Email Providers and Their IMAP Servers

Provider	IMAP server domain name
Gmail	*imap.gmail.com*
Outlook.com/Hotmail.com	*imap-mail.outlook.com*
Yahoo Mail	*imap.mail.yahoo.com*
AT&T	*imap.mail.att.net*
Comcast	*imap.comcast.net*
Verizon	*incoming.verizon.net*

IMAP – Logging in to the IMAP Server

```
print(imapObj.login('dfuentes@uccs.edu', PASSWORD))
# LOGIN completed.
```

- Remember to not save your passwords in your files:
  - *Use input()*
  - *Encrypt it*

IMAP – Searching for Email

- Once you're logged in, actually retrieving an email that you're interested in is a two-step process.
    1. *You must select a folder you want to search through.*
    2. *You must call the IMAPClient object's search() method, passing in a string of IMAP search keywords.*

IMAP – Searching for Email – Selecting a Folder

```
imapObj.select_folder('INBOX', readonly=True)
```

- Almost every account has an INBOX folder by default, but you can also get a list of folders by calling the IMAPClient object's list_folders() method. This returns a list of tuples. Each tuple contains information about a single folder.
- **If you do want emails to be marked as read when you fetch them, you will need to pass *readonly=False* to *select_folder()***

```
pprint.pprint(imapObj.list_folders())
# [(('\\HasNoChildren',), '/', u'Archive'),
#  (('\\HasNoChildren', '\\Drafts'), '/', u'Drafts'),
#  (('\\Marked', '\\HasNoChildren'), '/', u'INBOX'),
#  (('\\HasNoChildren', '\\Sent'), '/', u'Sent Items')]
```

IMAP – Searching for Email – Performing the Search

```
UIDs = imapObj.search(['SINCE', date(2019, 3, 22)])
print(UIDs) # UIDs: Unique IDs
# [7227, 7229, 7231, 7233, 7234, 7237, 7239, 7241, 7243, 7248]
```

- The argument to search() is sequence of one or more criteria items. Example values:

```
['UNSEEN']
['SMALLER', 500]
['NOT', 'DELETED']
['TEXT', 'foo bar', 'FLAGGED', 'SUBJECT', 'baz']
['SINCE', date(2005, 4, 3)]
```

- Full list of possible criteria items:
  - https://gist.github.com/martinrusev/6121028

IMAP – Searching for Email – Search criteria

Table 16-3: IMAP Search Keys

Search key	Meaning
`'ALL'`	Returns all messages in the folder. You may run in to `imaplib` size limits if you request all the messages in a large folder. See "Size Limits" on page 371.
`'BEFORE` *`date`*`'`, `'ON` *`date`*`'`, `'SINCE` *`date`*`'`	These three search keys return, respectively, messages that were received by the IMAP server before, on, or after the given *`date`*. The date must be formatted like `05-Jul-2015`. Also, while `'SINCE 05-Jul-2015'` will match messages on and after July 5, `'BEFORE 05-Jul-2015'` will match only messages before July 5 but not on July 5 itself.
`'SUBJECT` *`string`*`'`, `'BODY` *`string`*`'`, `'TEXT` *`string`*`'`	Returns messages where *`string`* is found in the subject, body, or either, respectively. If *`string`* has spaces in it, then enclose it with double quotes: `'TEXT "search with spaces"'`.

(continued)

Table 16-3 (continued)

Search key	Meaning
`'FROM string'`, `'TO string'`, `'CC string'`, `'BCC string'`	Returns all messages where *string* is found in the "from" emailaddress, "to" addresses, "cc" (carbon copy) addresses, or "bcc" (blind carbon copy) addresses, respectively. If there are multiple email addresses in *string*, then separate them with spaces and enclose them all with double quotes: `'CC "firstcc@example.com secondcc@example.com"'`.
`'SEEN'`, `'UNSEEN'`	Returns all messages with and without the *\Seen* flag, respectively. An email obtains the *\Seen* flag if it has been accessed with a `fetch()` method call (described later) or if it is clicked when you're checking your email in an email program or web browser. It's more common to say the email has been "read" rather than "seen," but they mean the same thing.
`'ANSWERED'`, `'UNANSWERED'`	Returns all messages with and without the *\Answered* flag, respectively. A message obtains the *\Answered* flag when it is replied to.

`'DELETED'`, `'UNDELETED'`	Returns all messages with and without the *\Deleted* flag, respectively. Email messages deleted with the `delete_messages()` method are given the *\Deleted* flag but are not permanently deleted until the `expunge()` method is called (see "Deleting Emails" on page 375). Note that some email providers, such as Gmail, automatically expunge emails.
`'DRAFT'`, `'UNDRAFT'`	Returns all messages with and without the *\Draft* flag, respectively. Draft messages are usually kept in a separate `Drafts` folder rather than in the `INBOX` folder.
`'FLAGGED'`, `'UNFLAGGED'`	Returns all messages with and without the *\Flagged* flag, respectively. This flag is usually used to mark email messages as "Important" or "Urgent."
`'LARGER` *`N`*`'`, `'SMALLER` *`N`*`'`	Returns all messages larger or smaller than *`N`* bytes, respectively.
`'NOT` *`search-key`*`'`	Returns the messages that *`search-key`* would *not* have returned.
`'OR` *`search-key1 search-key2`*`'`	Returns the messages that match *either* the first or second *`search-key`*.

`imapObj.search(['ALL'])` Returns every message in the currently selected folder.

`imapObj.search(['ON 05-Jul-2015'])` Returns every message sent on July 5, 2015.

`imapObj.search(['SINCE 01-Jan-2015', 'BEFORE 01-Feb-2015', 'UNSEEN'])` Returns every message sent in January 2015 that is unread. (Note that this means *on and after* January 1 and up to *but not including* February 1.)

`imapObj.search(['SINCE 01-Jan-2015', 'FROM alice@example.com'])` Returns every message from *alice@example.com* sent since the start of 2015.

`imapObj.search(['SINCE 01-Jan-2015', 'NOT FROM alice@example.com'])` Returns every message sent from everyone except *alice@example.com* since the start of 2015.

`imapObj.search(['OR FROM alice@example.com FROM bob@example.com'])` Returns every message ever sent from *alice@example.com* or *bob@example.com*.

`imapObj.search(['FROM alice@example.com', 'FROM bob@example.com'])` Trick example! This search will never return any messages, because messages must match *all* search keywords. Since there can be only one "from" address, it is impossible for a message to be from both *alice@example.com* and *bob@example.com*.

IMAP – Searching for Email – Size limits

- If your search matches a large number of email messages, Python might raise an exception that says imaplib.error: got more than 10000 bytes . When this happens, you will have to disconnect and reconnect to the IMAP server and try again.
- This limit is in place to prevent your Python programs from eating up too much memory. Unfortunately, the default size limit is often too small. You can change this limit from 10,000 bytes to 10,000,000 bytes by running this code:

```
import imaplib
imaplib._MAXLINE = 10000000
```

IMAP – Searching for Email – Fetching an Email and Marking It As Read

```
lastUID = UIDs[len(UIDs)-1] # One UID or a list of UIDs
rawMessages = imapObj.fetch(lastUID, ['BODY[]', 'FLAGS'])
print(rawMessages)
# defaultdict(<type 'dict'>, {7239: {'FLAGS': ('\\Recent',),
'SEQ': 2674, 'BODY[]': 'X-Antivirus: avast (VPS 19032304)\r\n...
```

- Once you have a list of UIDs, you can call the IMAPClient object's fetch() method to get the actual email content.
- The list of UIDs will be fetch() 's first argument. The second argument should be the list *['BODY[]']* , which tells fetch() to download all the body content for the emails specified in your UID list.
- As you can see, the message content in the 'BODY[]' key is pretty unintelligible. It's in a format called RFC 822, which is designed for IMAP servers to read. But you don't need to understand the RFC 822 format; pyzmail module will make sense of it for you.

IMAP – Searching for Email – Getting Email Addresses from a Raw Message

- The raw messages returned from the fetch() method still aren't very useful to people who just want to read their email. The **pyzmail module** parses these raw messages and returns them as PyzMessage objects, which make the subject, body, "To" field, "From" field, and other sections of the email easily accessible to your Python code.

IMAP – Searching for Email – Getting Email Addresses from a Raw Message

```
message = pyzmail.PyzMessage.factory(rawMessages[lastUID]['BODY[]'])
print(message.get_subject())
# Welcome to the Python course!!
print(message.get_addresses('from'))
# [(u'Spotify', 'hello@spotify.com')]
print(message.get_addresses('to'))
# [(u'dfuentes@uccs.edu', 'dfuentes@uccs.edu')]
print(message.get_addresses('cc'))
# []
```

IMAP – Searching for Email – Getting Email Addresses from a Raw Message

```
print(message.text_part != None)

# True

print(message.text_part.get_payload().decode(message.text_part.charset))

# Your Release Radar is here: Fresh tracks by your favorite artists,
including Why Don't We, on a personalized playlist that we ...

print(message.html_part != None)

# True

print(message.html_part.get_payload().decode(message.html_part.charset))

# <!DOCTYPE html><html xmlns="http://www.w3.org/1999/xhtml" style="margin:
0; padding: 0"><head># <meta http-equiv="Content-Type" content="text/html;
charset=utf-8"><meta name="viewport" ...
```

IMAP – Searching for Email – Getting Email Addresses from a Raw Message

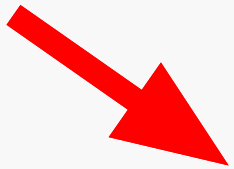
If the email doesn't have text part or html part, these would be False

```
print(message.text_part != None)
# True
print(message.text_part.get_payload().decode(message.text_part.charset))
# Your Release Radar is here: Fresh tracks by your favorite artists,
including Why Don't We, on a personalized playlist that we ...
print(message.html_part != None)
# True
print(message.html_part.get_payload().decode(message.html_part.charset))
# <!DOCTYPE html><html xmlns="http://www.w3.org/1999/xhtml" style="margin:
0; padding: 0"><head># <meta http-equiv="Content-Type" content="text/html;
charset=utf-8"><meta name="viewport" ...
```

IMAP – Searching for Email – Deleting emails

- To delete emails, pass a list of message UIDs to the IMAPClient object's delete_messages() method. This marks the emails with the \Deleted flag.
- Calling the expunge() method will permanently delete all emails with the \Deleted flag in the currently selected folder.

```
imapObj.select_folder('INBOX', readonly=False)
UIDs = imapObj.search(['ON 09-Jul-2015'])
print(UIDs)
# [40066]
print(imapObj.delete_messages(UIDs))
# {40066: ('\\Seen', '\\Deleted')}
print(imapObj.expunge())
# ('Success', [(5452, 'EXISTS')])
```



readonly=False so we can delete emails.

IMAP - Disconnecting from the IMAP Server

```
imapObj.logout()
```

- If your program runs for several minutes or more, the IMAP server may time out, or automatically disconnect. In this case, the next method call your program makes on the IMAPClient object will raise an exception.

SENDING TEXT MESSAGES WITH TWILIO

Sending Text Messages

- Most people are more likely to be near their phones than their computers, so text messages can be a more immediate and reliable way of sending notifications than email.
- Also, the short length of text messages makes it more likely that a person will get around to reading them.

Twilio

- Cloud communications platform for building SMS, Voice & Messaging applications on an API built for global scale.
  - *http://twilio.com/*
- We are going to use the **SMS gateway service**, which means it's a service that allows you to send text messages from your programs.
- With the free indefinite account you will be limited in how many texts you can send per month and the texts will be prefixed with the words *Sent from a Twilio trial account.*
- While in trial mode, you can only send messages to non-Twilio phone numbers you've verified with Twilio.
- If you prefer not to use Twilio, you can find alternative services by searching online for *free sms gateway*, *python sms api*, or even *twilio alternatives*.

Twilio – Register

- Go to http://twilio.com/ and fill out the sign-up form. Once you've signed up for a new account, you'll need to verify a mobile phone number that you want to send texts to. (This verification is necessary to prevent people from using the service to spam random phone numbers with text messages.)
- After receiving the text with the verification number, enter it into the Twilio website to prove that you own the mobile phone you are verifying. You will now be able to send texts to this phone number using the twilio module.
- Twilio provides your trial account with a phone number to use as the sender of text messages
- Finally, **from the Dashboard obtain your account SID and the auth (authentication) token**.

Twilio - Sending Text Messages

```
from twilio.rest import Client

accountSID = 'ACxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
authToken = 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
client = Client(accountSID, authToken)
myTwilioNumber = '+14434718468’
myCellPhone = '+17193528571’
message = client.messages \
                .create(
                     body="Join Earth's mightiest heroes. Like Kevin Bacon.",
                     from_=myTwilioNumber,
                     to=myCellPhone
                 )
```

Twilio – Checking messages information

```
print(message.to)            # +17193528571
print(message.from_)         # +14434718468
print(message.body)  # Sent from your Twilio trial account –
                     # Join Earth's mightiest heroes. Like Kevin Bacon.
print(message.status)         # queued
print(message.date_created) # 2019-03-24 01:24:36+00:00
print(message.date_sent == None)   # True
print(message.sid)            # SM5e2e8819c14142689e67891b392c62f8
time.sleep(10)
updatedMessage = client.messages.get(message.sid).fetch()
print(updatedMessage.status)        # delivered
print(updatedMessage.date_sent)    # 2019-03-24 01:24:36+00:00
```

Project proposals

- Everyone that has 15 points and has not talked with me yet means that we need to talk about the proposal. So, you can come to talk with me now.

Homework 9 – Exercises 1 and 2