Part 23
-
Send Mail

Version: 2020-02-20

Sending emails from the Raspberry Pi using msmtp

Looks like sSMTP is no longer maintained, and has been orphaned since 19th March 2019. MSMTP is the suggested replacement. So let's get replacing.

Install mSMTP

```
$ apt -y update
$ apt -y install msmtp msmtp-mta
```

Show version

```
$ msmtp –version

msmtp version 1.8.3
Platform: arm-unknown-linux-gnueabihf
TLS/SSL library: GnuTLS
Authentication library: GNU SASL
Supported authentication methods:
plain scram-sha-1 external gssapi cram-md5 digest-md5 login ntlm
IDN support: enabled
NLS: enabled, LOCALEDIR is /usr/share/locale
Keyring support: none
System configuration file name: /etc/msmtprc
User configuration file name: /root/.msmtprc

Copyright (C) 2019 Martin Lambers and others.
This is free software.  You may redistribute copies of it under the terms of
the GNU General Public License <http://www.gnu.org/licenses/gpl.html>.
There is NO WARRANTY, to the extent permitted by law.
```

Configure mSMTP to our smtp server

We need to setup a configuration file for msmtp to have the smtp details. The general msmtp config location is in `/etc/`. The name is msmtprc
Let's create the file and add the following configuration:

```
$ nano /etc/msmtprc

defaults
tls_trust_file /etc/ssl/certs/ca-certificates.crt
logfile /var/log/msmtp.log

# MyISP
account myisp
host smtp.myisp.com
port 25
auth plain
tls on
from mymail@myisp.com
user <mylogin>
password <mypassword>

account default : myisp
aliases /etc/aliases
```

Note: Change to meet the requirements of your isp

Note:

- The possible option for `auth` are `plain, scram-sha-1, external, gssapi, cram-md5, digest-md5, login, ntlm`

- If used, the these lines need to be come after the account settings.

```
account default : myisp
aliases /etc/aliases
```

- You can have multiple account settings.

The above is system-wide. You can create a user-specific msmtp config file also. This file is located in the home directory of the user. The name is `.msmtprc`. So for root this would be `/root/.msmtprc` and for pi `/home/pi/.msmtprc`
Make the `msmtprc` config file readable/writable by its owner

```
$ chmod 600 /etc/msmtprc
$ chmod 600 ~/.msmtprc
```

Aliases
As you saw, we also set an aliases path in the config file.

```
aliases /etc/aliases
```

Let's create this file for our users. This file should contain Linux usernames to email mappings like this.

```
$ nano /etc/aliases

root: root@myisp.com
pi: pi@gmail.com
default: mymail@myisp.com
```

Symbolic link
Create a symbolic link for msmtp in the path used by sendmail.

```
$ rm /usr/sbin/sendmail
$ ln -s /usr/bin/msmtp /usr/sbin/sendmail
```

Send an email
First create a text file we'll use to send mails

```
$ nano mail.txt

Subject: test email

hello world!
```

Save it. Now send a mail to root

```
$ sendmail -d -t root <mail.txt
```

The option `-d` activate the debugging which might help you to figure out what goes wrong when you mail does not arrive at its destination.

Extend the text file with To: and/or From: to control these fields.

```
To: myto@myisp.com
From: myfrom@domain.be
Subject: test email

hello world!
```

If you want to hide your password
Use `passwordeval` to hide and retrieve your password

```
defaults
tls_trust_file /etc/ssl/certs/ca-certificates.crt
logfile /var/log/msmtp.log

# MyISP
account myisp
host smtp.myisp.com
port 587
auth plain
tls on
from mymail@myisp.com
user <mylogin>
passwordeval 'gpg --quiet --for-your-eyes-only --no-tty --decrypt ~/.msmtp-
credentials.gpg'

account default : myisp
aliases /etc/aliases
```

then create a GPG encrypted password file,

If you have not use gpg before, first generate your keypair

```
$ gpg --full-generate-key
```

Follow what's on the screen.
If mailaddress is asked, use same as you would use below <user>@myisp.com

Next by command:

```
$ gpg --encrypt -o ~/.msmtp-credentials.gpg -r <user>@myisp.com -
```

The ending dash is not a typo, rather it causes `gpg` to use `stdin`. After running that snippet of code, type in your password, press enter, and press `Control-d` so `gpg` can encrypt your password.

Test the decryption

```
$ gpg --quiet --for-your-eyes-only --no-tty --decrypt ~/.mail/.msmtp-
credentials.gpg
```

You can use also the command mail

```
$ mail -t < mail.txt
```

Redirect emails from cron
The Cron utility emails the output and errors of an executed job to the appropriate Linux user unless you redirect these output to files or other streams.
You can use msmtp to receive emails from cron jobs to each users' preferred email ID through your preferred email provider.
Make sure msmtp is installed and configured, you have the aliases set-up and all is tested (see above)

Create a cron job which echos some text which causes an email to be sent.

```
$ crontab -e

* * * * *   echo "A message from Cron"
```

You'll have to wait for 60 seconds for the cron job to execute.
Have a look at your syslog or messages file to know what is happening

```
tail -f /var/log/syslog
```

Make sure to remove the cron job immediately else an email will be triggered every minute and your Email Service provider might suspend your account.

If you want to redirect all mails from cron to a specific mailaddress, add

```
MAILTO=cron.server@myisp.com
```

in the cron job list

```
MAILTO=cron.server@myisp.com
* * * * *   echo "A message from Cron"
```

Sending emails from the Raspberry Pi using Python smtplib

- **For a basic email**

  There's a native library in Python to send emails: `smtplib`. No need to install external librairies. To send a basic email (without subject line), with a Gmail address, the code is fairly simple:

```
import smtplib

server = smtplib.SMTP('smtp.gmail.com', 587)
server.starttls()
server.login("YOUR EMAIL ADDRESS", "YOUR PASSWORD")

msg = "YOUR MESSAGE!"
server.sendmail("YOUR EMAIL ADDRESS", "THE EMAIL ADDRESS TO SEND TO", msg)
server.quit()
```

  On line 3, it's the parameters for the Gmail server. First, the server location (or its ip address), then the port to use. If you have an email address from another service, like Yahoo for example, you have to find the corresponding information. On line 4, there's a security function, needed to connect to the Gmail server. It will protect your password. Don't forget to indicate your email address and your password on line 5. The *msg* variable will contain your message and the next line will send it!

- **For a more elaborate email**

  If you want to automatically send an email to the police for an interview each time there's a press release regarding drug trafficking, you need to send more professional looking emails!
  With the code below, you will send a clean email, with a sender, a receiver and a subject line.
  To do this, you need two more modules:

```
Python 2: email.MIMEMultipart and email.MIMEText
Python 3: email.mime.multipart and email.mime.text
```

  They are part of the basic Python librairies. No need to install them.

```
import smtplib
from email.mime.multipart import MIMEMultipart
from email.mime.text      import MIMEText

from = "YOUR ADDRESS"
to   = "ADDRESS YOU WANT TO SEND TO"
msg  = MIMEMultipart()
msg['From'] = from
msg['To']   = to
msg['Subject'] = "SUBJECT OF THE MAIL"

body = "YOUR MESSAGE HERE"
msg.attach(MIMEText(body, 'plain'))

server = smtplib.SMTP('smtp.gmail.com', 587)
server.starttls()
server.login(from, "YOUR PASSWORD")
text = msg.as_string()
server.sendmail(from, to, text)
server.quit()
```

- **To send an email with attachment**

  To have files attached to an email requires a more complicated code. In summary, the essential step is to convert the file into a Base64 before sending it. My code works for text files, pdf files, images, audio files and video files!

```
import smtplib
from email.mime.multipart import MIMEMultipart
from email.mime.text      import MIMEText
from email.mimi.base      import MIMEBase
from email                import encoders


from = "YOUR EMAIL"
to   = "EMAIL ADDRESS YOU SEND TO"

msg            = MIMEMultipart()
msg['From']    = from
msg['To']      = to
msg['Subject'] = "SUBJECT OF THE EMAIL"

body = "TEXT YOU WANT TO SEND"

msg.attach(MIMEText(body, 'plain'))

filename   = "NAME OF THE FILE WITH ITS EXTENSION"
attachment = open("PATH OF THE FILE", "rb")

part = MIMEBase('application', 'octet-stream')
part.set_payload((attachment).read())
encoders.encode_base64(part)
part.add_header('Content-Disposition', "attachment; filename= %s" %
filename)

msg.attach(part)

server = smtplib.SMTP('smtp.gmail.com', 587)
server.starttls()
server.login(from, "YOUR PASSWORD")
text = msg.as_string()
server.sendmail(from, to, text)
server.quit()
```

SendMail on Raspberry Pi

This is a generic procedure and also works on other Debian distributions. This is not for sending mail only but also for receiving. Receiving email on the Pi is a much more involved process, requiring your Pi to preform the role of a full mail server.
The ability to send mail across the internet is useful. It enables your scripts and applications to send you email about system events or sending data such as pictures from a webcam.

- **Install Sendmail**

  This procedure uses sendmail, the great grandaddy of all email programs. Install it now as follows.Update software lists and install:

  ```
  sudo apt -y update
  sudo apt –y install sendmail
  ```

  This will install sendmail and a few other packages, create the directory `/etc/mail`, and perform a few other tasks.

- **Configure Sendmail**

  Change to `/etc/mail` and edit the file `/etc/mail/submit.mc`, using an editor of your choice. (Do not edit the `submit.cf` file)

  ```
  cd /etc/mail
  nano submit.mc
  ```

  Add the three lines shown below in bold to the file. Put them just after the "`DOMAIN(`debian-msp')dnl`" line, as shown:

  ```
  define(`_USE_ETC_MAIL_')dnl
  include(`/usr/share/sendmail/cf/m4/cf.m4')dnl
  VERSIONID(`$Id: submit.mc, v 8.14.4-4 2013-02-11 11:12:33 cowboy Exp $')
  OSTYPE(`debian')dnl
  DOMAIN(`debian-msp')dnl
  define(`SMART_HOST',`[mail.telenet.be]')dnl
  FEATURE(`authinfo',`hash -o /etc/mail/authinfo.db')dnl
  MASQUERADE_AS(`engrie.be')dnl
  ```

  Here, `mail.telenet.be` is the mail server of my provider. My domain name is `engrie.be`. Email sent from the Pi will appear to come from an email address "`<user>@engrie.be`". Save the `submit.mc` file after making the above changes.

  Next, edit the `authinfo` file.

  ```
  nano authinfo
  ```

  Make it contain a single line, like this one:

  ```
  AuthInfo: "U:myusername@telenet.be" "P:mypassword" "M:PLAIN"
  ```

  `myusername@telenet.be` is my email address at the ISP. `mypassword` is the password to go with it. Enter the appropriate address and password for you. Sendmail will use this information to authenticate with your ISP every time it sends an email. It is a basic spam prevention measure implemented by many service providers.
  Save the `authinfo` file and use `makemap` to create the binary version. The following command creates a file called "`authinfo.db`", using the information you placed into `authinfo`.

  ```
  makemap hash /etc/mail/authinfo < /etc/mail/authinfo
  ```

Finally, edit your `/etc/hosts` file and add your domain to the line containing "raspberrypi". That is the same domain name you used in **`submit.mc`**, above (in my case, `engrie.be`). On a newly installed Raspbian Jessie, it looks like this:

```
127.0.1.1       raspberrypi
```

Change that to:

```
127.0.1.1       raspberrypi engrie.be
```

Now restart sendmail. This can take a couple of minutes sometimes, so be patient:

```
service sendmail restart
```

- **Send a Test Email**

  Send a test email to a mail account you own, eg gmail, hotmail, outlook or whatever:

  ```
  /usr/lib/sendmail -v someuser@hotmail.com
  ```

  Type some text

  ```
  test mail
  ```

  Hit `<ctrl-d>`

  This will print a load of stuff as it sends the mail, including hopefully a "`235 PLAIN authentication successful`" message. Check your mail account to see the mail came through. Check the "junk" mail folder too. It should not take more than a second or two, or maybe a few minutes at very busy times.

- **Fix Local Email**

  If the above test worked, Internet mail is working from your Pi. Great. Now, however, local mail fails. By "`local mail`", I mean email sent from one user on the Pi to another user, for example when `cron` sends a user the output from their scheduled jobs, or when the system sends diagnostic messages to the root user. It may not be important to you, but best make it work anyway. Proceed as follows.

  Tell sendmail not to authenticate for local mail. Edit the file `/etc/mail/access` and add this line:

  ```
  SRV_Features:127.0.0.1 A
  ```

  Save the file and type `make` to regenerate its binary version, `access.db`:

  ```
  cd /etc/mail
  nano access
  make
  Creating /etc/mail/relay-domains
  Optional file...
  Updating access_db ...
  The following file(s) have changed:
  ** ** You should issue `/etc/init.d/sendmail reload` ** **
  ```

  Restart sendmail:

  ```
  service sendmail restart
  ```

If you see messages complaining about "sasl2-bin" not being installed, and about enabling SASL2 support at a later date, they can be ignored for the purposes of this procedure.
As a test, try sending a local mail. Eg send a mail to the "pi" user.

```
/usr/lib/sendmail -v pi

Hello pi user from root
<ctrl-d>
```

Again, lots of diagnostic stuff is printed, indicating successful delivery, including something like "050 … Sent".
Check that user pi did indeed receive the mail. We could login as user pi and read the mail with an application (do that by all means), but just tailing the user's mail file is quicker:

```
tail /var/mail/pi
```

You will see the received mail, ie. lots of header information followed by a blank line and your actual message:

```
Hello pi user from root
```

- **Raspbian Wheezy or later only: Enable IPV6**

At around the end of March 2016, an update was made to Raspbian Wheezy (Debian 7) that will prevent local mail from working unless you have IPv6 enabled. This only affects users of Wheezy. IPv6 is enabled by default on Jessie (Debian 8), so it isn't a problem. However, Wheezy users might see errors like these after updating their systems with apt -y. Eg. local mail fails, even after making all of the configurations above:

```
sendmail -v pi
test
pi... Connecting to [127.0.0.1] via relay...
pi... Deferred: Connection refused by [127.0.0.1]
The "Connection refused" message indicates that the sendmail daemon is
not running.
```

Check it with:

```
ps -ef | grep sendmail
```

If it isn't running, attempt to start the daemon:

```
service sendmail restart
```

If that falls (check with `ps` again), look at the end of the file `/var/log/mail.log` for messages like these:

```
tail /var/log/maillog
tail /var/log/mail.log
Apr  3 14:52:07 raspberrypi sm-mta[3064]: NOQUEUE: SYSERR(root):
opendaemonsocket: daemon MTA-v6: can't create server SMTP socket: Address
family not supported by protocol
Apr  3 14:52:07 raspberrypi sm-mta[3064]: daemon MTA-v6: problem creating
SMTP socket
Apr  3 14:52:07 raspberrypi sm-mta[3064]: NOQUEUE: SYSERR(root):
opendaemonsocket: daemon MTA-v6: server SMTP socket wedged: exiting
```

The sendmail daemon is failing to start because it seems to require IPv6 to be running.
After much messing about, I found the easiest solution was just to enable IPv6.
Edit the file `/etc/modules`

```
vi /etc/modules
```

and add a single line to the end of it:

```
ipv6
```

and reboot the Pi:

```
shutdown -r 0
```

After the system reboots, local mail should be working and sendmail should be running correctly:

```
sendmail -v pi
test
<ctrl-d>
ps -ef | grep send
root      2381     1  0 00:21 ?        00:00:00 sendmail: MTA: accepting
connections
```

- **Send a Picture**

  It is quite easy to send email attachments from the command line or a script. One of the best ways is to use `mutt`. Install `mutt`:

  ```
  sudo apt -y install -y mutt
  ```

  Send an email. You don't need to be root for this, so type `<ctrl-d>` to go back to the non-root user. The "`echo`" bit here just sends an empty email, and the "`-a`" attaches the jpg file. On my Pi, there is an image called "`driveway.jpg`" installed by default. Email it as follows. Note the double minus (–) seperating the destination adddress from the other options:

  ```
  $ echo | mutt -a /usr/share/scratch/Media/Backgrounds/Outdoors/driveway.jpg -
  s "Photo" --someuser@hotmail.com
  ```

Sending Emails With Python

Python comes with the built-in smtplib module for sending emails using the Simple Mail Transfer Protocol (SMTP). smtplib uses the RFC 821 protocol for SMTP. The examples in this tutorial will use the Gmail SMTP server to send emails, but the same principles apply to other email services. Although the majority of email providers use the same connection ports as the ones in this tutorial, you can run a quick Google search to confirm yours.

To get started with this tutorial, set up a Gmail account for development, or set up an SMTP debugging server that discards emails you send and prints them to the command prompt instead. Both options are laid out for you below. A local SMTP debugging server can be useful for fixing any issues with email functionality and ensuring your email functions are bug-free before sending out any emails.

Option 1: Setting up a Gmail Account for Development

If you decide to use a Gmail account to send your emails, I highly recommend setting up a throwaway account for the development of your code. This is because you'll have to adjust your Gmail account's security settings to allow access from your Python code, and because there's a chance you might accidentally expose your login details.
A nice feature of Gmail is that you can use the + sign to add any modifiers to your email address, right before the @ sign. For example, mail sent to my+person1@gmail.com and my+person2@gmail.com will both arrive at my@gmail.com. When testing email functionality, you can use this to emulate multiple addresses that all point to the same inbox.

To set up a Gmail address for testing your code, do the following:

- Create a new Google account the normal way
- Turn *Allow less secure apps* to *ON*. Login with the account, select "Manage your Google Account" by right clicking on the account icon. Select "Security" from the left panel, srcoll down till you find "Less secure app access" and turn it on. Be aware that this makes it easier for others to gain access to your account.

If you don't want to lower the security settings of your Gmail account, check out Google's documentation on how to gain access credentials for your Python script, using the OAuth2 authorization framework.

Option 2: Setting up a Local SMTP Server

You can test email functionality by running a local SMTP debugging server, using the smtpd module that comes pre-installed with Python. Rather than sending emails to the specified address, it discards them and prints their content to the console. Running a local debugging server means it's not necessary to deal with encryption of messages or use credentials to log in to an email server.
You can start a local SMTP debugging server by typing the following in Command Prompt:

```
sudo python -m smtpd -c DebuggingServer -n localhost:1025
```

Any emails sent through this server will be discarded and shown in the terminal window as a bytes object for each line:

```
---------- MESSAGE FOLLOWS ----------
b'X-Peer: ::1'
b''
b'From: my@address.com'
b'To: your@address.com'
b'Subject: a local test mail'
b''
b'Hello there, here is a test email'
------------ END MESSAGE ------------
```

For the rest of the tutorial, I'll assume you're using a Gmail account, but if you're using a local debugging server, just make sure to use localhost as your SMTP server and use port 1025 rather than port 465 or 587. Besides this, you won't need to use login() or encrypt the communication using SSL/TLS.

Starting a Secure SMTP Connection
When you send emails through Python, you should make sure that your SMTP connection is encrypted, so that your message and login credentials are not easily accessed by others. SSL (Secure Sockets Layer) and TLS (Transport Layer Security) are two protocols that can be used to encrypt an SMTP connection. It's not necessary to use either of these when using a local debugging server.

There are two ways to start a secure connection with your email server:

- Start an SMTP connection that is secured from the beginning using SMTP_SSL().
- Start an unsecured SMTP connection that can then be encrypted using .starttls().

In both instances, Gmail will encrypt emails using TLS, as this is the more secure successor of SSL. As per Python's Security considerations, it is highly recommended that you use create_default_context() from the ssl module. This will load the system's trusted CA certificates, enable host name checking and certificate validation, and try to choose reasonably secure protocol and cipher settings.
If you want to check the encryption for an email in your Gmail inbox, go to *More → Show original* to see the encryption type listed under the *Received* header.
smtplib is Python's built-in module for sending emails to any Internet machine with an SMTP or ESMTP listener daemon.

I'll show you how to use SMTP_SSL() first, as it instantiates a connection that is secure from the outset and is slightly more concise than the .starttls() alternative. Keep in mind that Gmail requires that you connect to port 465 if using SMTP_SSL(), and to port 587 when using .starttls().

Option 1: Using SMTP_SSL()
The code example below creates a secure connection with Gmail's SMTP server, using the SMTP_SSL() of smtplib to initiate a TLS-encrypted connection. The default context of ssl validates the host name and its certificates and optimizes the security of the connection.

```
import smtplib, ssl

smtpserver = "smtp.gmail.com"
smtpport = 465  # For SSL
smtplogin = "mytestaccount@gmail.com"
smtppassword = "testpassword"

# Create a secure SSL context
context = ssl.create_default_context()

with smtplib.SMTP_SSL(smtpserver, smtpport, context=sslcontext) as server:
    server.login(smtplogin, smtppassword)
    # TODO: Send email here
```

Using `with smtplib.SMTP_SSL() as server` makes sure that the connection is automatically closed at the end of the indented code block. If port is zero, or not specified, .SMTP_SSL() will use the standard port for SMTP over SSL (port 465).

Option 2: Using .starttls()

Instead of using .SMTP_SSL() to create a connection that is secure, we can create an unsecured SMTP connection and encrypt it using .starttls().
To do this, create an instance of smtplib.SMTP, which encapsulates an SMTP connection and allows you access to its methods.
The code snippet below uses the construction `server = SMTP()`, rather than the format with `SMTP() as server` which we used in the previous example. To make sure that your code doesn't crash when something goes wrong, put your main code in a try block, and let an except block print any error messages to stdout:

```
import smtplib, ssl

smtpserver = "smtp.gmail.com"
smtpport = 587  # For SSL
smtplogin = "mytestaccount@gmail.com"
smtppassword = "testpassword"

# Create a secure SSL context
context = ssl.create_default_context()

# Try to log in to server and send email
try:
    server = smtplib.SMTP(smtpserver, smptport)
    server.starttls(context=sslcontext) # Secure the connection
    server.login(smtplogin, smtppassword)
    # TODO: Send email here

except Exception as e:
    # Print any error messages to stdout
    print(e)

finally:
    server.quit()
```

Sending Your Plain-text Email
After you initiated a secure SMTP connection using either of the above methods, you can send your email using .sendmail(), which pretty much does what it says:

```
server.sendmail(From, To, message)
```

So

```
From = "mytestaccount@gmail.com"
To = "your@gmail.com"
message = """\
Subject: Hi there

This message is sent from Python."""

# Send email here
```

The message string starts with `"Subject: Hi there"` followed by two newlines (\n). This ensures `Hi there` shows up as the subject of the email, and the text following the newlines will be treated as the message body. The code example below sends a plain-text email using SMTP_SSL():

```
import smtplib, ssl

smtpserver = "smtp.gmail.com"
smtpport = 465  # For SSL
smtplogin = "mytestaccount@gmail.com"
smtppassword = "testpassword"

From = "mytestaccount@gmail.com"
To = "your@gmail.com"
message = """\
Subject: Hi there

This message is sent from Python."""

# Create a secure SSL context
sslcontext = ssl.create_default_context()

with smtplib.SMTP_SSL(smtpserver, smtpport, context=sslcontext) as server:
    server.login(smtplogin, smtppassword)
    server.sendmail(From, To, message)
```

For comparison, here is a code example that sends a plain-text email over an SMTP connection secured with .starttls().

```
import smtplib, ssl

smtpserver = "smtp.gmail.com"
smtpport = 587  # For SSL
smtplogin = "mytestaccount@gmail.com"
smtppassword = "testpassword"

From = "mytestaccount@gmail.com"
To = "your@gmail.com"
message = """\
Subject: Hi there

This message is sent from Python."""
```

```
# Create a secure SSL context
sslcontext = ssl.create_default_context()

# Try to log in to server and send email
try:
    server = smtplib.SMTP(smtpserver, smtpport)
    server.starttls(context=sslcontext) # Secure the connection
    server.login(smtplogin, smtppassword)
    server.sendmail(From, To, message)

except Exception as e:
    # Print any error messages to stdout
    print(e)

finally:
    server.quit()
```

Sending Fancy Emails
Python's built-in email package allows you to structure more fancy emails, which can then be transferred with smtplib as you have done already.

If you want to format the text in your email (bold, *italics*, and so on), or if you want to add any images, hyperlinks, or responsive content, then HTML comes in very handy. Today's most common type of email is the MIME (Multipurpose Internet Mail Extensions) Multipart email, combining HTML and plain-text. MIME messages are handled by Python's email.mime module.

As not all email clients display HTML content by default, and some people choose only to receive plain-text emails for security reasons, it is important to include a plain-text alternative for HTML messages. As the email client will render the last multipart attachment first, make sure to add the HTML message after the plain-text version.
In the example below, MIMEText() objects will contain the HTML and plain-text versions of the message, and the MIMEMultipart("alternative") instance combines these into a single message with two alternative rendering options:

```
import smtplib, ssl
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart

sender_email = "my@gmail.com"
receiver_email = "your@gmail.com"
password = input("Type your password and press enter:")

message = MIMEMultipart("alternative")
message["Subject"] = "multipart test"
message["From"] = sender_email
message["To"] = receiver_email

# Create the plain-text and HTML version of your message
text = """\
Hi,
How are you?
Real Python has many great tutorials:
www.realpython.com"""
html = """\
<html>
  <body>
    <p>Hi,<br>
       How are you?<br>
       <a href="http://www.realpython.com">Real Python</a>
       has many great tutorials.
    </p>
  </body>
</html>
"""

# Turn these into plain/html MIMEText objects
part1 = MIMEText(text, "plain")
part2 = MIMEText(html, "html")

# Add HTML/plain-text parts to MIMEMultipart message
# The email client will try to render the last part first
message.attach(part1)
message.attach(part2)

# Create secure connection with server and send email
sslcontext = ssl.create_default_context()
with smtplib.SMTP_SSL("smtp.gmail.com", 465, context=sslcontext) as server:
    server.login(sender_email, password)
```

```
    server.sendmail(sender_email, receiver_email, message.as_string())
```

Adding Attachments

In order to send binary files to an email server that is designed to work with textual data, they need to be encoded before transport. This is most commonly done using base64, which encodes binary data into printable ASCII characters.

The code example below shows how to send an email with a PDF file as an attachment:

```
import email, smtplib, ssl

from email import encoders
from email.mime.base import MIMEBase
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText

subject = "An email with attachment from Python"
body = "This is an email with attachment sent from Python"
sender_email = "my@gmail.com"
receiver_email = "your@gmail.com"
password = input("Type your password and press enter:")

# Create a multipart message and set headers
message = MIMEMultipart()
message["From"] = sender_email
message["To"] = receiver_email
message["Subject"] = subject
message["Bcc"] = receiver_email  # Recommended for mass emails

# Add body to email
message.attach(MIMEText(body, "plain"))

filename = "document.pdf"  # In same directory as script

# Open PDF file in binary mode
with open(filename, "rb") as attachment:
    # Add file as application/octet-stream
    # Email client can usually download this automatically as attachment
    part = MIMEBase("application", "octet-stream")
    part.set_payload(attachment.read())

# Encode file in ASCII characters to send by email
encoders.encode_base64(part)

# Add header as key/value pair to attachment part
part.add_header(
    "Content-Disposition",
    f"attachment; filename= {filename}",
)

# Add attachment to message and convert message to string
message.attach(part)
text = message.as_string()

# Log in to server using secure context and send email
sslcontext = ssl.create_default_context()
with smtplib.SMTP_SSL("smtp.gmail.com", 465, context=sslcontext) as server:
    server.login(sender_email, password)
    server.sendmail(sender_email, receiver_email, text)
```

```
import smtplib, ssl

from email.mime.multipart import MIMEMultipart
from email.mime.text      import MIMEText


# email info
smtpserver            = "smtp.engrie.local"
smtpport              = 587
smtptls               = True
smtpCA                = False
smtplogin             = ""
smtppass              = ""
From                  = strScriptBase + "." + COMPUTERNAME + "@engrie.local"
To                    = "marc@engrie.local"


# setup ssl
# Create a secure SSL context
sslcontext = ssl.create_default_context()
if smtptls:
    try:
        ssl._create_unverified_https_context = ssl._create_unverified_context

    except AttributeError:
        # Legacy Python that doesn't verify HTTPS certificates by default
        pass


def sendmail(From, To, Subject, Body, Attach = ""):

    global smtpserver, smtpport, smtptls, smtpCA, smtplogin, smtppass, sslcontext
    global strLogging

    msg            = MIMEMultipart()
    msg['From']    = From
    msg['To']      = To
    msg['Subject'] = Subject

    msg.attach(MIMEText(Body, 'plain'))

    if Attach != "":
        attachment = open(Attach, "rb")
        part = MIMEBase('application', 'octet-stream')
        part.set_payload((attachment).read())
        encoders.encode_base64(part)
        part.add_header('Content-Disposition', "attachment; filename= %s" % Attach)
        msg.attach(part)

    try:
        server = smtplib.SMTP(smtpserver, smtpport)
        if smtptls and not smtpCA:
            server.starttls()           # Secure the connection
        elif smtptls and smtpCA:
            server.starttls(sslcontext) # Secure the connection
        if smtplogin != "":
            server.login(smtplogin, smtppass)
        text = msg.as_string()
        server.sendmail(From, To, text)

    except Exception as e:
        print(e)

    finally:
        server.quit()



sendmail(From, To, Subject, Body, Attach)
```