



Gunjan Khaitan
a month ago



Python Script To Send Email

Python Script to Send Emails

Hello everyone, A common task for system administrators and developers is to use scripts to send emails if an error occurs. So this video is useful to perform that kind of tasks.
In this video, I explained how to send email from sender to receiver id with the help of python.

How to send emails using Python

You probably found this tutorial because you want to send emails using Python. Perhaps you want to receive email reminders from your code, send a confirmation email to users when they create an account, or send emails to members of your organization to remind them to pay their dues. Sending emails manually is a time-consuming and error-prone task, but it’s easy to automate with Python.

In this tutorial you’ll learn how to:

- Set up a **secure connection** using `SMTP_SSL()` and `.starttls()`
- Use Python’s built-in `smtplib` library to send **basic emails**
- Send emails with **HTML content** and **attachments** using the `email` package
- Send multiple **personalized emails** using a CSV file with contact data
- Use the **Yagmail** package to send email through your Gmail account using only a few lines of code

You’ll find a few transactional email services at the end of this tutorial, which will come in useful when you want to send a large number of emails.

Table of Contents

- Getting Started
 - Option 1: Setting up a Gmail Account for Development
 - Option 2: Setting up a Local SMTP Server
- Sending a Plain-Text Email
 - Starting a Secure SMTP Connection
 - Sending Your Plain-text Email
- Sending Fancy Emails
 - Including HTML Content

- Adding Attachments Using the email Package
- Sending Multiple Personalized Emails
 - Make a CSV File With Relevant Personal Info
 - Loop Over Rows to Send Multiple Emails
 - Personalized Content
 - Code Example
- Yagmail
- Transactional Email Services
- Sendgrid Code Example
- Conclusion

Getting Started

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.

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. Also, I found that the inbox of my testing account rapidly filled up with test emails, which is reason enough to set up a new Gmail account for development.

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](#).
- Turn [Allow less secure apps to 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:

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

On Linux, use the same command preceded by `sudo`.

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.

Sending a Plain-Text Email

Before we dive into sending emails with HTML content and attachments, you'll learn to send plain-text emails using Python. These are emails that you could write up in a simple text editor. There's no fancy stuff like text formatting or hyperlinks. You'll learn that a bit later.

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.

`smtpplib` 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 `smtpplib` 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. Make sure to fill in your own email address instead of `my@gmail.com`:

```
import smtpplib, ssl

port = 465 # For SSL
password = input("Type your password and press enter: ")

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

with smtpplib.SMTP_SSL("smtp.gmail.com", port, context=context) as server:
    server.login("my@gmail.com", password)
    # TODO: Send email here
```

Using `with smtpplib.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).

It's not safe practice to store your email password in your code, especially if you intend to share it with others. Instead, use `input()` to let the user type in their password when running the script, as in the example above. If you don't want your password to show on your screen when you type it, you can import the `getpass` module and use `.getpass()` instead for blind input of your password.

Option 2: Using `.starttls()`

Instead of using `.SMTP_SSL()` to create a connection that is secure from the outset, we can create an unsecured SMTP connection and encrypt it using `.starttls()`.

To do this, create an instance of `smtpplib.SMTP`, which encapsulates an SMTP connection and allows you access to its methods. I recommend defining your SMTP server and port at the beginning of your script to configure them easily.

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 smtpplib, ssl

smtp_server = "smtp.gmail.com"
port = 587 # For starttls
sender_email = "my@gmail.com"
password = input("Type your password and press enter: ")
```

```

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

# Try to log in to server and send email
try:
    server = smtplib.SMTP(smtp_server,port)
    server.ehlo() # Can be omitted
    server.starttls(context=context) # Secure the connection
    server.ehlo() # Can be omitted
    server.login(sender_email, password)
    # TODO: Send email here
except Exception as e:
    # Print any error messages to stdout
    print(e)
finally:
    server.quit()

```

To identify yourself to the server, `.helo()` (SMTP) or `.ehlo()` (ESMTP) should be called after creating an `.SMTP()` object, and again after `.starttls()`. This function is implicitly called by `.starttls()` and `.sendmail()` if needed, so unless you want to check the SMTP service extensions of the server, it is not necessary to use `.helo()` or `.ehlo()` explicitly.

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 on the tin:

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

I recommend defining the email addresses and message content at the top of your script, after the imports, so you can change them easily:

```

sender_email = "my@gmail.com"
receiver_email = "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

port = 465 # For SSL
smtp_server = "smtp.gmail.com"
sender_email = "my@gmail.com" # Enter your address
receiver_email = "your@gmail.com" # Enter receiver address
password = input("Type your password and press enter: ")
message = """\
Subject: Hi there

This message is sent from Python.""

context = ssl.create_default_context()
with smtplib.SMTP_SSL(smtp_server, port, context=context) as server:
    server.login(sender_email, password)
    server.sendmail(sender_email, receiver_email, message)

```

For comparison, here is a code example that sends a plain-text email over an SMTP connection secured with `.starttls()`. The `server.ehlo()` lines may be omitted, as they are called implicitly by `.starttls()` and `.sendmail()`, if required:

```

import smtplib, ssl

port = 587 # For starttls
smtp_server = "smtp.gmail.com"

```

```

smtp_server = smtp.gmail.com
sender_email = "my@gmail.com"
receiver_email = "your@gmail.com"
password = input("Type your password and press enter:")
message = ""\
Subject: Hi there

This message is sent from Python."""

context = ssl.create_default_context()
with smtplib.SMTP(smtp_server, port) as server:
    server.ehlo() # Can be omitted
    server.starttls(context=context)
    server.ehlo() # Can be omitted
    server.login(sender_email, password)
    server.sendmail(sender_email, receiver_email, message)

```

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. Below, you'll learn how use the `email` package to send emails with HTML content and attachments.

Including HTML Content

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. For a detailed description, check [the documentation](#).

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, our `MIMEText()` objects will contain the HTML and plain-text versions of our 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
context = ssl.create_default_context()
with smtplib.SMTP_SSL("smtp.gmail.com", 465, context=context) as server:
    server.login(sender_email, password)
    server.sendmail(
        sender_email, receiver_email, message.as_string()
    )

```

In this example, you first define the plain-text and HTML message as string literals, and then store them as [plain/html MIMEText](#) objects. These can then be added in this order to the [MIMEMultipart\("alternative"\)](#) message and sent through your secure connection with the email server. Remember to add the HTML message after the plain-text alternative, as email clients will try to render the last subpart first.

Adding Attachments Using the [email](#) Package

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()

```



```

text = message.as_string()

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

```

The `MIMEultipart()` message accepts parameters in the form of [RFC5233](#)-style key/value pairs, which are stored in a dictionary and passed to the `.add_header method` of the `Message` base class.

Check out the [documentation](#) for Python's `email.mime` module to learn more about using MIME classes.

Sending Multiple Personalized Emails

Imagine you want to send emails to members of your organization, to remind them to pay their contribution fees. Or maybe you want to send students in your class personalized emails with the grades for their recent assignment. These tasks are a breeze in Python.

Make a CSV File With Relevant Personal Info

An easy starting point for sending multiple personalized emails is to create a CSV (comma-separated values) file that contains all the required personal information. (Make sure not to share other people's private information without their consent.) A CSV file can be thought of as a simple table, where the first line often contains the column headers.

Below are the contents of the file `contacts_file.csv`, which I saved in the same folder as my Python code. It contains the names, addresses, and grades for a set of fictional people. I used `my+modifier@gmail.com` constructions to make sure all emails end up in my own inbox, which in this example is [my@gmail.com](#):

```

name,email,grade
Ron Obvious,my+ovious@gmail.com,B+
Killer Rabbit of Caerbannog,my+rabbit@gmail.com,A
Brian Cohen,my+brian@gmail.com,C

```

When creating a CSV file, make sure to separate your values by a comma, without any surrounding whitespaces.

Loop Over Rows to Send Multiple Emails

The code example below shows you how to open a CSV file and loop over its lines of content (skipping the header row). To make sure that the code works correctly before you send emails to all your contacts, I've printed `Sending email to ...` for each contact, which we can later replace with functionality that actually sends out emails:

```

import csv

with open("contacts_file.csv") as file:
    reader = csv.reader(file)
    next(reader) # Skip header row
    for name, email, grade in reader:
        print(f"Sending email to {name}")
        # Send email here

```

In the example above, using `with open(filename) as file:` makes sure that your file closes at the end of the code block. `csv.reader()` makes it easy to read a CSV file line by line and extract its values. The `next(reader)` line skips the header row, so that the following line `for name, email, grade in reader:` splits subsequent rows at each comma, and stores the resulting values in the strings `name`, `email` and `grade` for the current contact.

If the values in your CSV file contain whitespaces on either or both sides, you can remove them using the `.strip()` method.

Personalized Content

You can put personalized content in a message by using `str.format()` to fill in curly-bracket placeholders. For example, `"hi {name}, you {result} your assignment".format(name="John", result="passed")` will give you `"hi John, you passed your assignment"`.

As of Python 3.6, string formatting can be done more elegantly using f-strings, but these require the placeholders to be defined before the f-string itself. In order to define the email message at the beginning of the script, and fill in placeholders for each contact when looping over the CSV file, the older `.format()` method is used.

With this in mind, you can set up a general message body, with placeholders that can be tailored to individuals.

Code Example

Below is a code example that sends out emails to all contacts in the CSV file. The code uses the `email.mime` module to create a multipart message body, and the `smtplib` module to send the email.

The following code example lets you send personalized emails to multiple contacts. It loops over a CSV file with `name,email,grade` for each contact.

The general message is defined in the beginning of the script, and for each contact in the CSV file its `{name}` and `{grade}` placeholders are filled in, and a personalized email is sent out through a secure connection with the Gmail server, as you saw before:

```
import csv, smtplib, ssl

message = """Subject: Your grade

Hi {name}, your grade is {grade}"""
from_address = "my@gmail.com"
password = input("Type your password and press enter: ")

context = ssl.create_default_context()
with smtplib.SMTP_SSL("smtp.gmail.com", 465, context=context) as server:
    server.login(from_address, password)
    with open("contacts_file.csv") as file:
        reader = csv.reader(file)
        next(reader) # Skip header row
        for name, email, grade in reader:
            server.sendmail(
                from_address,
                email,
                message.format(name=name, grade=grade),
            )
```

Yagmail

There are multiple libraries designed to make sending emails easier, such as [Envelopes](#), [Flanker](#) and [Yagmail](#). Yagmail is designed to work specifically with Gmail, and it greatly simplifies the process of sending emails through a friendly API, as you can see in the code example below:

```
import yagmail

receiver = "your@gmail.com"
body = "Hello there from Yagmail"
filename = "document.pdf"

yag = yagmail.SMTP("my@gmail.com")
yag.send(
    to=receiver,
    subject="Yagmail test with attachment",
    contents=body,
    attachments=filename,
)
```

When setting up Yagmail, you can add your Gmail validations to the keyring of your OS, as described in [the documentation](#). If you don't do this, Yagmail will prompt you to enter your password when required and store it in the keyring automatically.

Transactional Email Services

If you plan to send a large volume of emails, want to see email statistics, and want to ensure reliable delivery, it may be worth looking into transactional email services. Although all of the following services have paid plans for sending large volumes of emails, they also come with a free plan so you can try them out. Some of these free plans are valid indefinitely and may be sufficient for your email needs. You can run a [Google search](#) to see which provider best fits your needs, or just try out a few of the free plans to see which API you like working with most.

Sendgrid Code Example

Here's a code example for sending emails with [Sendgrid](#) to give you a flavor of how to use a transactional email service with Python:

```
import os
import sendgrid
from sendgrid.helpers.mail import Content, Email, Mail

sg = sendgrid.SendGridAPIClient(
    apikey=os.environ.get("SENDGRID_API_KEY")
)
from_email = Email("my@gmail.com")
to_email = Email("your@gmail.com")
```



```
subject = "A test email from Sendgrid"
content = Content(
    "text/plain", "Here's a test email sent through Python"
)
mail = Mail(from_email, subject, to_email, content)
response = sg.client.mail.send.post(request_body=mail.get())

# The statements below can be included for debugging purposes
print(response.status_code)
print(response.body)
print(response.headers)
```

To run this code, you must first:

- [Sign up for a \(free\) Sendgrid account](#)
- [Request an API key](#) for user validation
- Add your API key by typing `setx SENDGRID_API_KEY "YOUR_API_KEY"` in Command Prompt (to store this API key permanently) or `set SENDGRID_API_KEY YOUR_API_KEY` to store it only for the current client session

More information on how to set up Sendgrid for Mac and Windows can be found in the repository’s README on [Github](#).

Conclusion

You can now start a secure SMTP connection and send multiple personalized emails to the people in your contacts list!

You’ve learned how to send an HTML email with a plain-text alternative and attach files to your emails. The [Yagmail](#) package simplifies all these tasks when you’re using a Gmail account. If you plan to send large volumes of email, it is worth looking into transactional email services.

 1

 79.45 GEEK





Write a comment

usman imtiaz
14 days ago
it show error when i run sendgrid code: TypeError: Object of type Email is not JSON serializable

No more data :)



django

Python Django Full Stack Web Developer

3,569 students enrolled





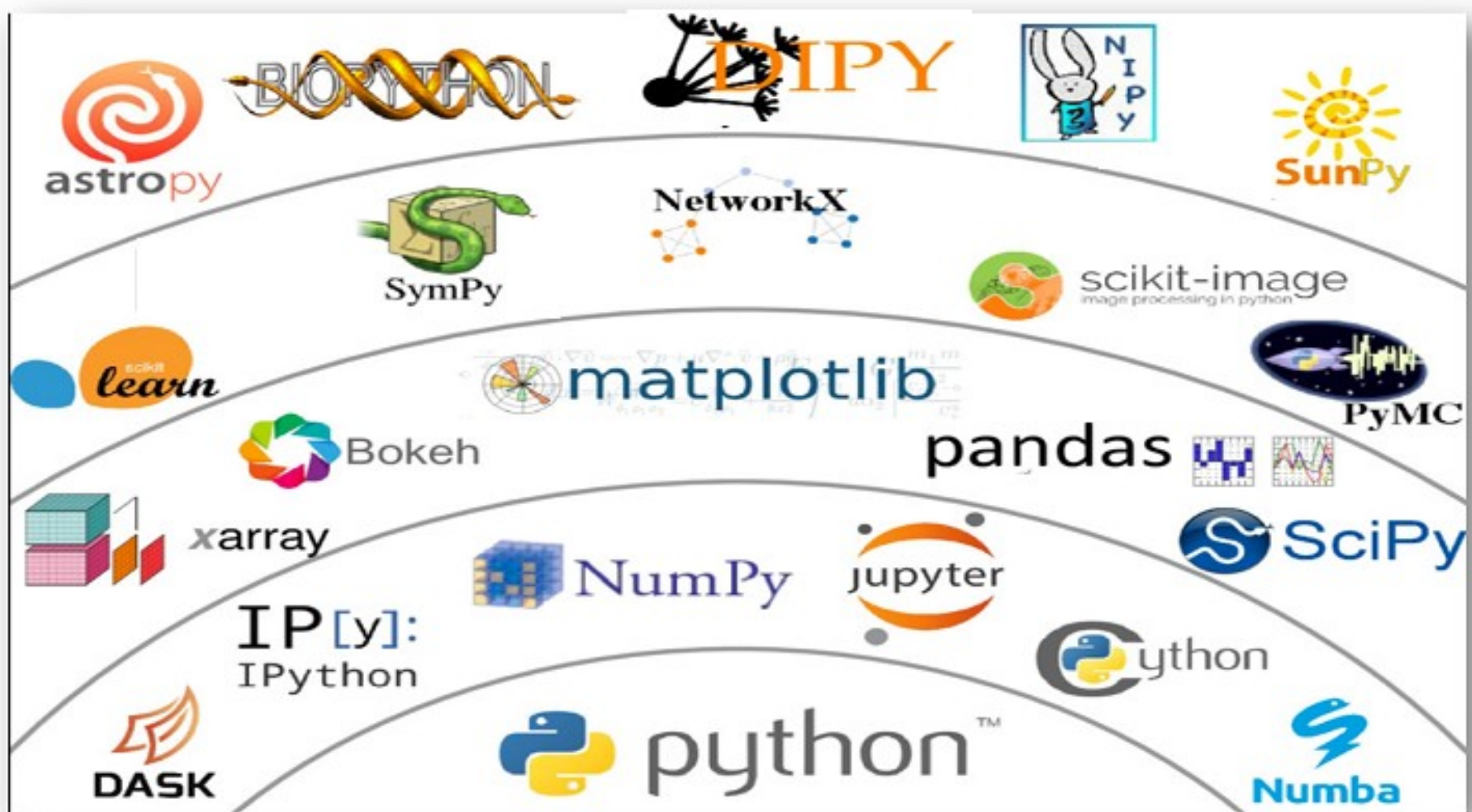
Complete Python Course: Zero to Mastery

739 students enrolled




Getting Started Web Development Tools and Resources 2020

2,674 students enrolled




Python AI and Machine Learning for Production & Development

9,053 students enrolled





Login

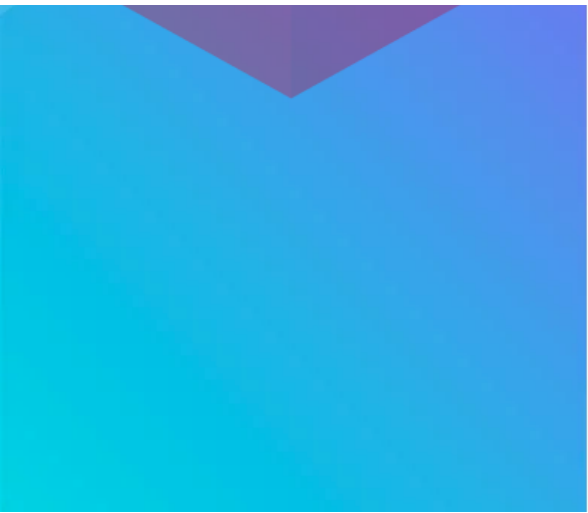
Username

 Type your username

Password

 Type your password





Forgot password?

LOGIN

Or Sign Up Using

f

t

G





Reactjs Tutorial
3,885 likes

Like Page



Send Message

Python Tutorial - Learn Python for Machine Learning and Web Develop...

Web Design and Development Company

0.10 GEEK

Top 10 Python Frameworks for Web Development In 2019

Learn Flask - Python Web Development - Flask Crash Course For Begin...

11.75 GEEK

eCommerce Web Development Company

2.60 GEEK

Game Development with Python: Snake Game