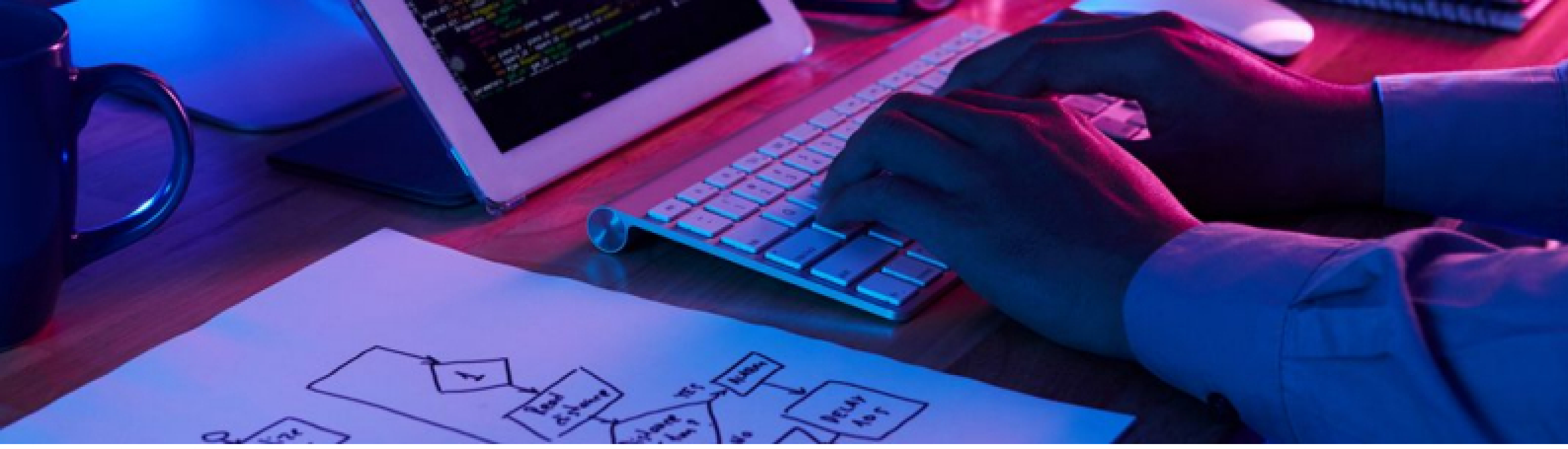

Automating Real-World Tasks with Python

Course 6 of 6 in Google IT Automation with Python Specialization.

bangkit

Automating Real-World Tasks with Python

Final Capstone Project

- This last course will bring the opportunity to put all new knowledge into action. With a bunch of small projects based on real-life scenarios, use Python to write the automation to solve the challenges. Throughout the course, it will give an introduction to the tools and techniques necessary for solving each of the projects.
- The last course is composed of **readings** (without videos) that guide the path of using Python to solve a real-life problem, along with labs to put the knowledge into action.

Manipulating Images

- Many developers write a Python module then publish it in **PyPi**, Python Package Index.
- This course modules are about **transforming** and **converting images** with Python Imaging Library (**PIL**). There's a package called **Pillow** which is a current up-to-date fork of PIL (hasn't been updated since 2009).

Install Python Imaging Library (PIL)

```
# install from linux dist package
$ sudo apt install python3-pil

# install from pip (pip3) package
manager
$ pip3 install pillow

# test by import PIL in Python console
$ python
>>> import PIL
```

* Python Package Index, PyPi https://pypi.org

Application Programming Interface (API)

- Application Programming Interface (API) helps different pieces of software talk to each other. Modules or libraries provide **APIs** in the form of **external** or **public** functions, classes, and methods that other code can use without having to create a lot of repeated code.
- An API is sort of like a promise. Even the library's internal code changes, expect the function to keep accepting the same parameters and returning the same results.
- When a library author needs to make a **breaking changes** to an API, they need to have a plan in place for communicating that change.

Application Programming Interface (API)

- In general a good API should be descriptive. A well-designed API will follow patterns and **naming conventions**. This means that the functions, classes, and methods should have names that help users understand what to expect from them.
- When the name isn't enough, the next step is to refer to its documentation.
- In Python, if the code of module or library is documented using docstrings. Use **`help()`** function to describe a function. Eg: `help(PIL)`

Manipulating Images

- Let's take a sample code from Pillow.
- This piece of code is pretty straightforward. Even you haven't seen this library before, it's easy to guess that the code opens an image called bridge.jpg, rotates it 45 degrees, and then shows on the screen.

Pillow API

```
from PIL import Image
im = Image.open("bridge.jpg")
im.rotate(45).show()
```

bangkit

Manipulating Images

- When using PIL, it typically creates Image objects that hold the data associated with the images to be processed.
- With these objects, do operations by calling different methods that either **return a new image object** or **modify the data in the image**.
- And then it ends up saving the result in a different file.

Resize and Save Image

```
from PIL import Image
im = Image.open("example.jpg")
new_im = im.resize((640,480))
new_im.save("resized.jpg")
```

return a new object

tuple as parameter

Combine Rotate, Resize, and Save Image

```
im = Image.open("example.jpg")
im.rotate(180).resize((640,480)).save("new.jpg")
```

Module 1 Project

Module 1

- To complete this module, install pillow with pip3, then write a script that:
  - Able to be executed, set shebang `#!/usr/bin/env python`
  - Can list a directory, then iterate every file on directory
  - Use PIL to read the image file, currently with .tiff format, 192x192px, rotate 90 deg anti-clockwise
  - Use PIL to resize to 128x128px, rotate 90 deg, save as .jpeg format on predefined directory
- Lastly, be prepared by saving a copy of the script, as all the scripts (and techniques) will be combined in the last module.

Interacting with Web Services

Interacting with Web Services

- A **web application** is an application that interacts with the user over HTTP (HyperText Transfer Protocol), which is the protocol of the world-wide web. Lots of web applications also have APIs that can be used from the scripts.
- Web applications that have an API also known as **web services**. Instead of browsing to a web page to type and click around, use a program to send a message known as an **API call** to the web service.
- The part of the program that listens on the network for API calls is called an **API endpoint**.

Data Serialization

- Data **serialization** is the process of taking an in-memory data structure, like a Python object, and turning it into something that can be stored on disk or transmitted across network.
- Later, the file can be read or the network transmission can be received by another program and turned back into an object again. Turning the serialized object back into an in-memory object is called **deserialization**.
- In a previous course, we took a list of lists in memory and wrote it to disk as a Comma-Separated Value (CSV). This is an example of serialization.

Data Serialization Formats

- Most common data serialization formats:
  - JSON (JavaScript Object Notation) is the serialization format used the most in the course. Use the **json** module to convert dictionary or list of dictionaries to JSON format.
  - YAML (Yet Another Markup Language) has a lot in common with JSON. Both formats can be easily understood by a human when looking at the contents. Use the **yaml** module.
- Some other common ones like Python pickle, Protocol Buffers, or the eXtensible Markup Language (XML).

Data Serialization with JSON

- A number of web services sends messages back & forth using JSON.
- JSON supports a few elements of different data types: strings, numbers, objects (like Python dictionaries), arrays.
- With json module, use **`dumps()`** method to serialize Python objects and return a string. Use **`loads()`** method to deserialize JSON into basic Python objects, by parse a string.

JSON Serialization

```
>>> import json
>>> people = [{"name": "Sabrina",
"username": "sgreen"}, {"name":
"Eli", "username": "ejones"}]
>>>
>>> people_json = json.dumps(people)
>>> print(people_json)
[{"name": "Sabrina", "username":
"sgreen"}, {"name": "Eli",
"username": "ejones"}]
```

Python **Requests**

- The Python **Requests** library makes it super easy to write programs that send and receive HTTP.
- Here's the example of using Requests to make a **HTTP GET** request for **URL** with name google.com and use protocol https. The result is an object of type `requests.Response`. The `Response.text` is the HTML.

Python Requests Library

```
>>> import requests
>>> response =
requests.get('https://google.com')
>>> print(response.text[:81])
<!doctype html><html itemscope=""
itemtype="http://schema.org/WebPage"
lang="id">

>>> response.ok
True
>>> response.status_code
200
```

Python Requests

- HTTP supports several HTTP methods, like GET, POST, PUT, and DELETE.
- The **HTTP GET** method of course, retrieves or gets the resource specified in the URL.
- A GET request can have **parameters** as a form of data serialization. When data becomes complex, it is hard to represent it using query strings.

HTTP GET parameters

```
>>> p = {'q': 'bangkit 2020'}
>>> response =
requests.get('https://google.com',
params=p)
>>> response.request.url
'https://www.google.com/?q=bangkit+2020'
```

bangkit

Python Requests

- An alternative of represent query strings is using the **HTTP POST** method. This method sends or posts data to a web service. Eg: fill a web form and press submit button. It will send the data using POST method.
- With Python Requests, POST request very similar to GET request. Only change params attribute with **data** attribute.

HTTP Post data

```
>>> p = {"name": "bangkit", "year":
2020}
>>> response =
requests.post("https://example.com",
data=p)

>>> response.request.url
'https://example.com/'
>>> response.request.body
'name=bangkit&year=2020'
```

bangkit

Python Requests

- Application Programming Interfaces (APIs). Manipulating Images. Module 1 Project.
- As mentioned before, it's very common to send and receive data specifically in JSON format. With Requests module it's simply to change the parameter to **json**.

HTTP POST JSON

```
>>> p = {"name": "bangkit", "year":
2020}
>>> response =
requests.post("https://example.com",
json=p)

>>> response.request.url
'https://example.com/'
>>> response.request.body
b'{"name": "bangkit", "year": 2020}'
```

Django Web Framework

- The lab project at the of this module will feature a simple web application created using **Django**, a full-stack web framework written in Python.
- Django contains libraries that will help to write app's code, store & retrieve data, receive & response web requests.
- Alternative Python-based web frameworks: Flask, Bottle, CherryPy, CubicWeb, and so on.

Module 2 Project

Module 2

- To complete this module, use the web server with external IP address, submit JSON to the REST endpoint and preview the result
- Then write a script that:
    - Able to be executed, set shebang `#!/usr/bin/env python`
    - Can list a directory, then iterate every file on directory
    - Parse the text file as the input of dictionary (later as JSON format)
    - Use Python requests to send JSON with no error, check the results on the web as before
- Lastly, be prepared by saving a copy of the script, as all the scripts (and techniques) will be combined in the last module.

Automatic Output Generation

bangkit

Sending Emails from Python

- The Simple Mail Transfer Protocol (SMTP) and Multipurpose Internet Mail Extensions (MIME) standards define how email messages are constructed.
- Python has a built-in email module, which helps to easily construct an email message.
- **From**, **To**, and **Subject** are examples of email header fields.

Python EmailMessage

```
>>> from email.message import EmailMessage
>>>
>>> message = EmailMessage()
>>> message['From'] = 'auto@example.com'
>>> message['To'] = 'user@example.com'
>>> message['Subject'] = 'Greetings'
>>>
>>> print(message)
From: automation@example.com
To: user@example.com
Subject: Greetings
```

Sending Emails from Python

- To continue previous email message construction, add message body, which is the main content of the message.
- The `set_content()` method also automatically added a couple headers, like MIME-Version.

Email Body

```
>>> body = """Hey there!
...
... I'm learning to send emails using
Python!"""
>>> message.set_content(body)
>>> print(message)
From: automation@example.com
To: user@example.com
Subject: Greetings
MIME-Version: 1.0
Content-Type: text/plain;
charset="utf-8"
Content-Transfer-Encoding: 7bit

Hey there!

I’m learning to send emails using
Python!
```

Sending Emails from Python

- Email messages are made up completely of strings. When an attachment is added to email, whatever type the attachment happens to be, it's encoded as some form of text (serialized).
- Label the attachments with a **MIME type** and **subtype** in order for the recipient understand what to do with the attachments.

Add Email Attachments

```
>>> from os import path
>>> from mimetypes import guess_type
>>> attachment_path = "/tmp/example.png"
>>> attachment_filename =
path.basename(attachment_path)
>>> mime_type, _ =
guess_type(attachment_path)
>>> print(mime_type)
image/png
>>> mime_type, mime_subtype =
mime_type.split('/', 1)
>>> with open(attachment_path, 'rb') as ap:
...     message.add_attachment(
...         ap.read(),
...         maintype=mime_type,
...         subtype=mime_subtype,
...         filename=attachment_filename)
...
>>>
```

bangkit

Sending Emails from Python

- To send emails the computers use the Simple Mail Transfer Protocol (SMTP).
- Python has a built-in smtplib module, which helps to easily send an email message.
- As an example, let's connect to localhost of Linux host which is already installed and configured as the SMTP server.

Python smtplib

```
>>> import smtplib
>>> mail_server = smtplib.SMTP('localhost')
>>> mail_server.set_debuglevel(1)
>>> mail_server.send_message(message)
>>> mail_server.quit()
```

Generating PDFs with Python

- There's a few tools in Python to generate PDFs. The course employs ReportLab, as it has a lot of different features for creating PDF documents.
- The examples in the course mostly use high-level classes and methods in the Page Layout and Typography Using Scripts (**platypus**).

Generating PDF with ReportLab

```
>>> from reportlab.platypus import
SimpleDocTemplate, Paragraph
>>> from reportlab.lib.styles import
getSampleStyleSheet
>>>
>>> report =
SimpleDocTemplate("/tmp/report.pdf")
>>> styles = getSampleStyleSheet()
>>> report_title = Paragraph("Inventory
of Fruit", styles["h1"])
>>> report.build([report_title])
```

Automatic Output Generation

- To continue previous report that only contains Paragraph, let's add **Table** to the PDF report.
- To make a Table object, convert the existing fruit dictionary to **list-of-lists** or sometimes called a two-dimensional array.
- Call `build()` method with list of report_title and report_table.

Add Table to PDF Report

```
>>> fruit = {"figs": 1, "apples": 2,
"durians": 3}
>>> table_data = [[k ,v] for k, v in
fruit.items()]
>>> print(table_data)
[['figs', 1], ['apples', 2], ['durians', 3]]
>>>
>>> from reportlab.platypus import Table
>>> report_table = Table(data=table_data)
>>> report.build([report_title,
report_table])
```

Module 3 Project

Module 3

- To complete this module, send email from example script then check the email on the web mail with external IP address.
- Next, modify the example script with new sender email address and new row of table data, then check the email result on the web mail.
- Also modify the car script with 2 new calculations from JSON data. Use previous reports function to generate PDF and use emails function to send the report. Make sure to check the email and PDF report on the web mail.
- Lastly, be prepared by saving a copy of the script, as all the scripts (and techniques) will be combined in the last module.

Putting It All Together

Putting It All Together

- In the past few modules, there are various scripts that have been built with the objectives of:
  - How to modify images using the Python Image Library.
  - How to interact with web service using Python requests and sending data in JSON format.
  - How to generate PDF files and send emails with PDFs as an attachment.
- For the final project in this course, all the techniques and concepts above will be combined to build a solution to a complex IT task.

Final Course Project

Final Project

- To complete this app, list and iterate files from directory of images & texts.
- Write a "changeImage" script that:
  - Able to be executed, set shebang `#!/usr/bin/env python`
  - Read images with PIL, resize and save to JPEG format.
- Execute "example_upload" script and check the result on the web server with external IP address and path URL "/media/images/"
- Write a "supplier_image_upload" script that customize the example upload script with the data to be listed and iterated from images directory.

Final Project (cont.)

- Test the available REST endpoint with sample JSON data and check the result on the web server with external IP address.
- Write a "run" script that:
    - Able to be executed, set shebang `#!/usr/bin/env python`
    - List and iterate all files from text directory, parse the data with predefined format and type of the data to construct the dictionary (later as JSON format).
    - Use Python requests to send JSON with no error, check the results on the web as before.

Final Project (cont.)

- Write "emails", "reports", "report_email" scripts that:
    - List & iterate all files from text directory, parse the data, format PDF report as defined. Output the PDF into specified file name & directory.
    - Send the generated PDF report as email attachment.
    - The "emails" and "reports" scripts similar with the script from previous module project.

Final Project (cont.)

- Write "health_check" script that:
    - Get system data with Python library (eg. shutil, psutil).
        - Refer to the documentation to get CPU, Disk and RAM data.
    - Construct email with subject depends on the predefined conditions.
        - Use /etc/hosts file to verify the localhost existence.
    - Install linux package called **stress**, setup multiple connection to the server to run the script with activated stress test
    - Check the email result on the web mail with external IP address and path URL "/webmail"