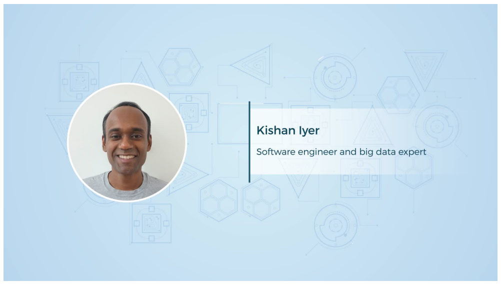
Python Requests: HTTP Requests with Python

Learners can explore how to use the Python Request package which has simplified the task of constructing HTTP requests in this 16-video lab course, which explores different types of HTTP requests, and examines several ways to handle responses to those requests. Begin by learning how to use the Python request package to make a GET request for data from a server. Then you will observe how to construct a POST request to submit data to a host, and how to send it to a URL. Continue by learning how to use a HEAD request to check the resource information before downloading it by using GET, and how to examine request and response headers. Next, learners will examine a PUT request which has the same effect whether one makes the request once or multiple times, and which is used to overwrite an existing resource. You will learn to use DELETE requests. Finally, you will learn to address responses to requests in both JSON formatted or images.

Table of Contents

1. Video: Course Overview (it_pywhpydj_01_enus_01)



Objectives

- *discover the key concepts covered in this course*

[Video description begins] *Topic title: Course Overview*. [Video description ends]

Hi, and welcome to this course, 'Making HTTP Requests with Python'. My name is Kishan Iyer, and I will be your instructor for this course.

[Video description begins] *Your host for this session is Kishan Iyer. He is a Software engineer and big data expert*. [Video description ends]

A little about myself first. I have a master's degree in computer science from Columbia University and have previously worked in companies such as Deutsche Bank and WebMD in New York. I presently work for Loonycorn, a studio for high-quality video content.

One of the fundamental tasks in any web application is to communicate data using HTTP, and the way to initiate communication with a remote server is by means of an HTTP request. Not too long ago, making such request involved a lot of setup work, from establishing and keeping alive HTTP connections to formatting and encoding query strings.

However, all of this is abstracted from the developer if they use the Python Requests package. The package advertises itself as HTTP for humans, and the sheer simplicity of using this package illustrates that this statement is definitely on point.

This course is purely hands-on, and involves a number of different labs, where we try out the different types of HTTP requests and explore different ways of handling the responses to those requests. We start off with the basic GET request, which is a way to ask a server for data, and then move on to POST requests, where we submit data to a host.

We then use PUT, HEAD, and DELETE requests, which have very specific use cases, and then get into the handling of different types of responses to those requests, whether the responses are JSON formatted or images. By the end of this course, you will know exactly how to use the Python Request package to retrieve and to submit data using HTTP. You will then be in a position to implement all that you have learned in your very own web application.

2. Video: Installing the Requests Package (it_pywhpydj_01_enus_02)

```
~>
~>ls
```

Objectives

- *install the Python Requests package and set up a workspace*

[Video description begins] *Topic Title: Installing the Requests Package. Your host for this session is Kishan Iyer.* [Video description ends]

In this course, we will be taking a look at the request package for Python. This package greatly simplifies the task of constructing HTTP requests.

[Video description begins] *A shell window appears on screen. The command prompt is: ~>.* [Video description ends]

An HTTP request can be thought of as a packet of information, which is sent from one computer to another, often over the Internet, in order to request some information, or to submit some data. By using the Python request package, such HTTP request can be made by simply specifying a URL endpoint, along with the parameters for the request.

All of the other details such as establishing and maintaining the connection to the endpoints is handled by the request package itself. To begin this course, I'm going to create a workspace directory within my file system. For this, I have brought up the shell on my MacOS and I'm first going to run an ls from my home directory.

[Video description begins] *He enters the ls command* . [Video description ends]

Within this directory, there are all of the default subdirectories, but what you will observe is that there is one for anaconda3.

[Video description begins] *A list of default sub directories appears. Some of these are: names: Applications, Documents, Downloads, Anaconda 3, etc.* [Video description ends]

Anaconda is a platform for both Python and R, and is very popular among data science professionals.

[Video description begins] *He highlights the anaconda 3 sub directory.* [Video description ends]

This includes a number of different packages and tools which are used in the data science field, and also includes a Python installation. The code for this course will be developed in Python 3, and it is required that you have this version of Python installed on your system along with jupyter notebooks, which is a commonly used Python IDE. We move along now to creating a work space directory, for which I will cd into my Desktop, and then from here, we can run an ls once more to view its contents.

[Video description begins] *He enters the command cd Desktop. The command prompt changes to ~ / Desktop>* [Video description ends]

[Video description begins] *He enters the ls command. It yields no output.* [Video description ends]

So currently I have nothing here, but I will now go ahead and create a new directory, which I shall call Requests. So I run an ls once more to confirm the creation of the directory, following which we can cd into it, and from here, we can begin writing the code in order to demonstrate the capabilities of the request package, and for that, I'm going to bring up the jupyter notebook IDE.

[Video description begins] *He enters the command mk dir Requests. He enters the ls command. It yields no output.* [Video description ends]

[Video description begins] *He enters the following command: cd Requests/.* [Video description ends]

[Video description begins] *The command prompt changes to: ~ / Desktop/Requests>. The following command appears: jupyter notebook.* [Video description ends]

As touched upon, jupyter notebooks are a very popular IDE for Python developers, which is widely used in the fields of data science and machine learning, and one reason for its widespread popularity is that it provides a REPL environment. REPL is short for Read, Eval, Print, Loop, and this is something which allows users to enter a few lines of code, execute it, and then view the results of that execution immediately before proceeding with the development of their programs. It is required that you have jupyter notebooks installed already, and this comes as part of the Anaconda installation. So by just running jupyter notebook from the command line this will bring up a web application and in my case this is running on port 8889 of my localhost.

[Video description begins] *He presses Enter. Several lines of output appear.* [Video description ends]

[Video description begins] *He highlights the following url in the output: http://localhost:8889/?token=. This is followed by an alphanumeric value.* [Video description ends]

So the jupyter notebook IDE is one which is available as a web application, and is compatible with all of the widely used browsers, such as Chrome, Firefox, and Safari. Once you run the jupyter notebook command from your shell, it is likely that the app has been brought up in your default browser. But in case it is not, you can copy over this URL, bring up the browser of your choice, and then paste this address in your address bar.

[Video description begins] *He copies the URL.* [Video description ends]

You can then navigate to it, and it should bring up the home page of the jupyter notebook IDE, just like what you see in the screen in front of you.

[Video description begins] *A Google chrome window appears in incognito mode. He pastes the URL in the address bar.* [Video description ends]

This view will list down all of the contents of the request directory from which you ran the jupyter notebook command, and in my case since the directory was empty there is nothing to view.

[Video description begins] *A jupyter notebook IDE Home page opens in the browser window. The title of the page is Home. The address bar has the following URL: localhost:8889/tree. Following buttons are present on the top right corner: Quit and Logout. It has three tabs: Files, Running, and Clusters. Currently, the Files tab is selected. Following buttons are present on the right corner of the tab: Upload , New and Refresh. A directory with the following buttons is present below the tab: Name, Last Modified, and File size. A message appears in the directory. It reads: The notebook is empty.* [Video description ends]

[Video description begins] *He highlights the following message: The notebook list is empty.* [Video description ends]

We can, however, fix that by creating a jupyter notebook. A jupyter notebook is something you can think of as an interactive Python source file, and in order to create such a notebook, you can navigate over to the new button in the top-right, and then click the option to create a new Python 3 notebook.

[Video description begins] *He clicks the New button on the top right corner. A drop down list appears. It has two categories. Notebook and Other. Python 3 is present under the Notebook category. The following options are present under the Other category: Text File, Folder, and Terminal. He selects Python 3 from the Notebook category.* [Video description ends]

If you have multiple versions of Python on your system, you may see each of those versions listed here. But for this course, we will be sticking with Python 3. Once you do that, a new notebook file will be created and this will automatically be brought up.

[Video description begins] *A new notebook file titled Untitled opens. It has Logout button on the top right corner. A menu bar is present with the following options: File, Edit, View, Insert, etc. A toolbar with the following buttons is present below the menu bar: Save, Cut, Copy, Run, etc. A code cell is present in the work area. It has the following label: In[]:. Currently, the code cell is blank.* [Video description ends]

You can see at the top here that the default name for this notebook has been set to Untitled, and before we proceed with coding, we will just assign it a new name.

[Video description begins] *He selects the title Untitled.* [Video description ends]

In this particular video, we will be seeing how we can create a GET request using the request package. So I'll change the name to reflect that by clicking on the name here, and then using this interface, we'll set it a new name of GET.

[Video description begins] *He clicks on the title. A pop up window with the title Rename Notebook appears. It has the following heading: Enter a new notebook name. An input field is present below the heading. He enters the text GET in the input field. The following buttons are present at the bottom: Cancel and Rename.* [Video description ends]

Once the rename button has been hit, the new name has been assigned to the notebook, and at this point, we can begin coding up our application.

[Video description begins] *He clicks the Rename button. The pop up window closes. The title of the jupyter notebook changes to GET. Currently, GET is highlighted.* [Video description ends]

To begin though, we need to install the request package, and the recommended way to install Python packages is by making use of the Python package manager called pip.

[Video description begins] *Several code cells appear. He enters the following code in the first code cell. !pip install requests --upgrade. Currently, the code is selected.* [Video description ends]

In order to install the request package, we will need to run the command pip install requests, and to execute such a command from a jupyter notebook, it needs to be preceded with an exclamation point. Also if you have the request package already and you wish to upgrade the existing installation, you can run this command with the upgrade flag. To execute this command from a jupyter notebook cell, there are a few options. You can hit the Run button in the top menu.

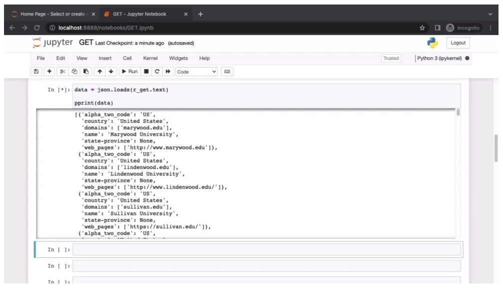
[Video description begins] *He highlights the Run button.* [Video description ends]

Alternatively, if you're on a macOS, you can use the keyboard shortcuts, Ctrl+Enter and Shift+Enter, where you'll first need to select the cell and then make use of the keyboard shortcuts. So I have made use of the keyboard shortcut now, which has executed the cell and the request package has been successfully installed.

[Video description begins] *He runs the code. Several lines of output appear. The last line of the output reads: Successfully installed requests-2.22.0. He highlights the last line.* [Video description ends]

At the time of this recording, the latest version of this request package is 2.22.0. So now that we have successfully performed the installation of the request package, it is time to begin coding up our first notebook, where we will be performing GET requests. That is something which will occur in the next video.

3. Video: A Basic GET Request (it_pywhpydj_01_enus_03)


Home Page - Select or create
GET - Jupyter Notebook
localhost:8888/notebooks/GET.ipynb
Incognito
jupyter GET Last Checkpoint: a minute ago (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3 (ipykernel)
Run
Code

```
In [*]: data = json.loads(r_get.text)

        pprint(data)
```

```
[{'alpha_two_code': 'US',
  'country': 'United States',
  'domains': ['marywood.edu'],
  'name': 'Marywood University',
  'state-province': None,
  'web_pages': ['http://www.marywood.edu']},
 {'alpha_two_code': 'US',
  'country': 'United States',
  'domains': ['lindenwood.edu'],
  'name': 'Lindenwood University',
  'state-province': None,
  'web_pages': ['http://www.lindenwood.edu/']},
 {'alpha_two_code': 'US',
  'country': 'United States',
  'domains': ['sullivan.edu'],
  'name': 'Sullivan University',
  'state-province': None,
  'web_pages': ['https://sullivan.edu/']},
 {'alpha_two_code': 'US',
```

```
In [ ]:
In [ ]:
In [ ]:
```

Objectives

- *make a GET request and explore the response object returned, which includes a status code and headers*

[Video description begins] *Topic title: A Basic GET Request. Your host for this session is Kishan Iyer.* [Video description ends]

So far, we have made use of Python's Request module in order to issue an HTTP GET Request and we have glimpsed into the response, which is returned to us. What we have not explored in the response so far is the raw data or the raw text and this can be accessed by means of the text field in the response object, r_get.text. But as implied by the name, this data will be in the text or string format, but it will be a JSON string.

[Video description begins] *A jupyter notebook titled GET is open in a browser window. It has a menu bar with the following options: File, Edit, View, Insert, etc. There is a tool bar with the following buttons: Save, Add, Cut, Run, etc. In the center pane, a set of code cells is present. The following code is present in code cell 1: !pip install requests --upgrade. Output for the code is present below the code cell. It's last line reads: Successfully installed requests-2.22.0.* [Video description ends] [Video description begins] *In code cell 2, he imports three lines of code. Line 1 is: import requests. Line 2 is: from pprint import pprint. Line 3 is: import json.* [Video description ends]

To translate it into a Python dictionary, which will be easy for us to view, we make use of the JSON modules 'loads' function.

This will return to us the payload in the response object in the form of a dictionary, and if there are several objects in the response, this would be in the form of a list of dictionaries, as we will soon see, which we will print out using the 'pprint' function. And this is what the data looks like. You'll observe that it is in fact an array of JSON objects, where each object represents an individual university in the United States.

The 'json.loads' function has translated this into a list of dictionaries, of course, the exact contents of the data will depend on the endpoint to which we have submitted the request and also the parameters, which we have passed along.

[Video description begins] *In code cell 3, he imports the following code: print(request.__version__).* [Video description ends]

[Video description begins] *An output appears below the third code cell. It reads: 2.22.0* [Video description ends]

In this case, the first element in the array points to Marywood University, followed by Lindenwood and Sullivan universities. As we scroll along the details for a whole host of universities can be seen, and you'll note that this is in fact a rather long list. Given that this is a list of dictionary objects we are now working with, we can perform various Python operations on it in order to access its data.

[Video description begins] *In code cell 4, he imports the following code: print(request.__copyright__).* [Video description ends] [Video description begins] *The following output appears below the fourth code cell: Copyright 2019 Kenneth Reitz.* [Video description ends]

For example, when we apply the length function, we get back a total of 2181, which points to the number of universities in the response. The data variable is of course a list, which means that we can access individual elements by making use of the appropriate index. Here when we take a look at the element at index three or the fourth element, what we get back are details of the Florida State College at Jacksonville. Moving along to access a specific attribute or field in that element.

[Video description begins] *In code cell 5, he imports the following code: r_get = requests.get('https: //www.metaweather.com/api/location/2487956/2018/11/28/').* [Video description ends]

Let's just say to access the value of the name property, we just specify name as the key. And sure enough, it is 'Florida State College at Jacksonville,' which shows up.

So by translating a JSON response into Python objects, it becomes easier for us to process it. Let's move along then, and take a look at some of the other fields in the response. For example, there is an 'is_redirect' property, which returns a Boolean value conveying whether our HTTP request was redirected to a different endpoint. A value of 'False' indicates that there was no redirection involved. And it was the endpoint which processed the request directly. Alright then! It's time for us to move along to a slightly different example. More specifically a different endpoint to which we will submit an HTTP GET request.

And to explore that, I'm first going to pull up a new tab in the browser. And from here we'll navigate over to swapi.dev.

[Video description begins] *In code cell 6, he imports the following code: r_get.status_code.* [Video description ends]

You'll note that this page points to a Star Wars API.

[Video description begins] *The following output value appears below the sixth code cell: 200.* [Video description ends]

This is a great resource for us to get familiar with issuing GET requests and processing responses. [Video description begins] *He imports the following code in code cell 7: type(r_get).* [Video description ends] And scrolling along, we see an example of such a request and response. This page includes a little form where we can point to a specific endpoint to which a request can be sent and then hit the request button in order to issue a get request to that endpoint.

[Video description begins] *The following output appears below the seventh code cell: requests.models.Response.* [Video description ends]

The URL for the endpoint begins with https://swapi.dev/api/ and then by default it is 'people/1/'. Let's issue a request to that endpoint and this is the response which we get back. So this is a single JSON object pointing to the character of Luke Skywalker. So the endpoint here is 'people/1'.

[Video description begins] *In code cell 8, he imports the following code: type(r_get.headers).* [Video description ends]

And this is a typical format applied with APIs, where you will have the type of object, in this case people, then a slash, followed by the ID of that object.

[Video description begins] *The following output appears below the eighth code cell: requests.structures.CaseInsensitiveDict.* [Video description ends]

Similarly when we submit a request for planet #3, so the endpoint is 'planets/3'.

[Video description begins] *He imports the following code in code cell 9: pprint(r_get.headers).* [Video description ends]

[Video description begins] *As an output, a set of content appears below the ninth code cell. It includes: 'Content-Language', 'Content-Type', 'Allow', and 'Content-Length'.* [Video description ends]

When we submit the request, the details of the planet Yavin 4 are returned. At the server end, Yavin 4 is planet #3, which is why we've been able to fetch the corresponding data via a request submitted to this endpoint. Well, we now have an idea of the data available in the Star Wars API, it's now time for us to make use of the Python request package in order to get this same data.

[Video description begins] *In the output, he highlights the following text: 'Allow' : 'GET, HEAD, OPTIONS'*. [Video description ends]

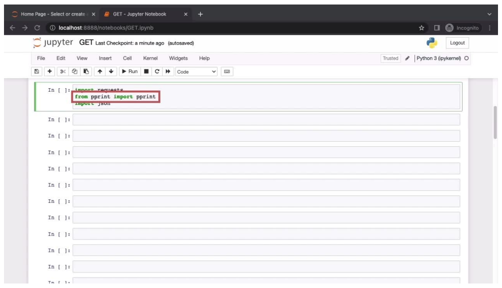
Returning them to a Jupiter notebook, well, let's issue a GET request once again.

This time to the endpoint 'swapi.dev/api/planet/3'. Sure enough, we get back a response object, we start off as we did before by examining the status code, again we get back a value of 200. So the request has been submitted and processed by the server. Let's take a look at the headers that are returned. Once again we are notified that it is an 'nginx' server to which we submitted the request. The content type and the response is once again 'application/json.' And while there are a few additional headers, from what we saw before, one of the interesting ones here is the 'Allow' header.

This points to the type of HTTP requests, which this endpoint is capable of processing, and the value of 'GET, HEAD and OPTIONS' conveys that it is these three types of HTTP requests which we can issue. This means that if you were to submit a post request to that same endpoint, it would not be handled. Let's move along then and take a look at the text Field in the response. This time we do not make use of the 'json.loads' method. So the contents will not be translated into Python objects, and we also don't apply the pretty print function. So this is what the raw data looks like.

You'll observe that the data is exactly what we saw earlier on in the 'swapi' website. And of course, in order to access specific fields in this response, we can just translate it into a dictionary by making use of json.loads. Now that we know how to issue HTTP get requests, in the next video, we will take a closer look at passing parameters along with the request in a little more detail.

4. Video: Exploring an HTTP Response Containing JSON Data (it_pywhpydj_01_enus_04)


Home Page - Select or create
GET - Jupyter Notebook
localhost:8888/notebooks/GET.ipynb
Incognito
jupyter GET Last Checkpoint: a minute ago (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3 (ipykernel)
Run
Code

```
In [ ]: import requests
        from pprint import pprint
        import json
```

Objectives

- *parse a response body containing JSON-formatted data*

[Video description begins] *Topic title: Exploring an HTTP Response Containing JSON Data. Your host for this session is Kishan Iyer.* [Video description ends]

Now that we have installed the request package, it is time for us to make use of it. I have now moved on to the next cell in this notebook and we'll start off by importing three different modules, which we will use in this demo.

[Video description begins] *A jupyter notebook titled GET is open in a browser window. It has a menu bar with the following options: File, Edit, View, Insert, etc. There is a tool bar with the following icons: Save, Add, Cut, Run, etc. In the center pane, a set of code cells is present. Code cell 10 contains two lines of code. Line 1 is: data = json.loads(r_get.text). Line 2 is: pprint(data).* [Video description ends]

At the very top is the requests module. We will be using this in order to issue get requests and then process the responses that are returned to us. We will be issuing requests to two different endpoints.

One of which will give us the data for various universities and then another one to a Star Wars API. The next module which we import is 'pprint,' from the 'pprint' package. This will allow us to format the responses which come to us into a form that is a little more easy to read than the raw data. And the raw data itself, well, at least most of it, will be in the JSON format.

[Video description begins] *An output appears below code cell 10. It includes the following details: air pressure, applicable date, created, humidity, id, max temp, min temp, visibility, etc.* [Video description ends]

And in order to process that we import the JSON module as well, the default format for data that is communicated via HTTP is JSON, and this is a very handy package in order to work with such data. Let's move along and take a look at the version of the requests module we'll be working with.

So we can access this via the Dunder version.

[Video description begins] *In the output, he highlights the following fields: 'applicable_date': '2018-11-28', 'max_temp': 16.335, 'min_temp': 12.97, and 'weather_state_name': 'Heavy Rain'*. [Video description ends]

Dunder field in the request module, when we print this out, well, this confirms that it's version 2.22.0, which we have installed. Other details of the request module may also be accessed. For example, the copyright information is available in the copyright field and as of this recording, this is what the copyright points to. And now we move on to the GET request itself.

[Video description begins] *A banner appears. It displays the following URL for Where On Earth Identifier (WOEID): http://www.woeidlookup.com/.* [Video description ends]

We can issue such a request by invoking the GET function in the request module [Video description begins] *In code cell 11, he imports the following code: r_get.is_redirect.* [Video description ends] and the argument here is the endpoint to which the request is submitted.

In this case, the URL which I have used is http://universities.hipolabs.com, and more specifically, its search endpoint. Beyond that we place in a question mark, which serves as a separator between the endpoint to which the request is submitted and then the arguments or parameters, which are passed along.

[Video description begins] *An output appears below the code cell. It reads: False.* [Video description ends]

In this case, the parameter is "country=United+States," which means that we are issuing a request for various universities in the United States as supplied by http://universities.hipolabs.com. When we issue a GET request in this manner, what we should get back in return is a response, which we cast into the variable, 'r_get.'

[Video description begins] *He opens a new browser window. The URL is: https://swapi.co/.* [Video description ends]

Let's run this cell then and it looks like there was no error when we invoke the request.

[Video description begins] *The page is titled SWAPI, The Star Wars API. The menu bar has the following options: Home, About, and Documentation.* [Video description ends]

So far so good, then. It's time for us to take a look at the various fields that are part of the response object, starting with the status code, which could point us towards various outcomes.

[Video description begins] *He scrolls down the page. An input field is present for URL: https://swapi.co/api/. Currently, the field contains the following request: people/1/. A request button is present next to the field. Below these, there is a section titled Result. It has an Editor area with a set of code lines. It contains the following details: name, height, mass, hair color, skin color, etc.* [Video description ends]

A value of 200 which is returned points to a status of OK, which means that the request was in the correct form and it has been accepted.

Generally, any status code in the 200s points to a correct request, which has been processed by the server, status codes in the 400s or 500s may point to errors. Let's move along then, and take a look at the type of the response object for which we apply the Python type function, and you'll observe that this is of type "requests.models.Response".

[Video description begins] *In the editor area, he highlights the name: "Luke Skywalker".* [Video description ends]

[Video description begins] *In the input field, he enters the following request: planets/3.* [Video description ends]

There are various fields in the response object beyond just the data which is returned to us. While the data will give us a list of US universities, there are other fields in the response, such as the headers, which we can also process.

[Video description begins] *He clicks the request button. The result for the request appears in the Editor area. He highlights the Name "Yavin IV".* [Video description ends]

First, let's take a look at the type of the header field in the response, and you'll note that this is a case insensitive dictionary.

[Video description begins] *In the result, he highlights the following text: "climate": "temperate, tropical".* [Video description ends]

[Video description begins] *He highlights the following text: "population": "1000".* [Video description ends]

The headers in a response object will give us metadata corresponding to the response, which can help us decide how exactly to process the response in the first place, before taking a look at the entire contents.

[Video description begins] *He shifts back to the jupyter notebook page.* [Video description ends]

Let's go ahead and take a look at the headers then, for which we make use of the 'pprint' module. And what is returned to us is the dictionary, or more specifically the case insensitive dictionary containing the headers.

[Video description begins] *He imports the following code in code cell 12: r_get = request.get('http://swapi.co/api/planets/3').* [Video description ends]

What are the interesting bits of information in here?

Well, first of all, the server property points to the fact that the endpoint to which we submitted our request is running an 'nginx' server and specifically version 1.14.0 of that server. Other important fields in the response include the content type, which points to the type of the payload or data in the response, which we will get to in a little bit.

[Video description begins] *The following code appears in code cell 13: r_get.status_code.* [Video description ends]

[Video description begins] *The following output value appears below the code cell: 200.* [Video description ends]

This is of type JSON, or in HTTP speak, 'application/json'.

[Video description begins] *In code cell 14, he imports the following code: pprint(r_get.headers).* [Video description ends]

Another important field in the headers is the content length, which in this case points to about 406 kilobytes of data in the payload.

[Video description begins] *A set of content appears below the code cell. It includes: 'Content-Type', 'Allow', 'Content-Encoding', etc. He points at the Allow content. It has the following values: 'GET, HEAD, OPTIONS'.* [Video description ends]

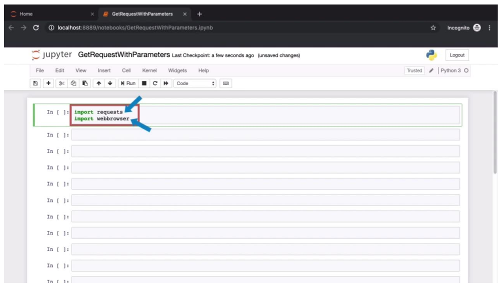
So you now have an idea of what the headers in a response object might contain.

[Video description begins] *In code cell 15, he imports the following code: r_get.text. An output appears below the code cell. It displays details for planet Yavin IV. It includes "name", "rotational_period", "diameter", etc* [Video description ends]

And, these can allow us to decide how exactly to process the data itself. Keep in mind that what is sent in the headers of the response is determined entirely by the server. If you submit your request to another server, you may get additional fields and the contents themselves may be a little different.

So now that we have issued an HTTP GET request, in the next video, let's take a look at the payload or the raw data in the response.

5. Video: Including Parameters in a GET Request (it_pywhpydj_01_enus_05)



```
import requests
import webbrowser
```

Objectives

- *invoke a GET request that includes parameters*

[Video description begins] *Topic title: Including Parameters in a GET Request. Your host for this session is Kishan Iyer.* [Video description ends]

So far in this course, we have taken a look at the creation of a simple GetRequest in order to get information from specific websites.

[Video description begins] *A jupyter notebook titled GetRequestWithParameters is open in a browser window. It has a menu bar with the following options: File, Edit, View, Insert, etc. There is a tool bar with the following icons: Save, Add, Cut, Run, etc. In the center pane, a few code cells are present.* [Video description ends]

We will now extend the GetRequest in this demo by passing along parameters along with the URL in order to get more specific information from the website to which we are making the GetRequest. I have now created a new jupyter notebook for this particular demo, and this I have named GetRequestWithParameters. Once the notebook has been created, we can begin writing our code, and once again, we begin by importing the requests module, and then, along with that, we will also import the web browser module which we will use in order to access a web browser instance from within the jupyter notebook.

[Video description begins] *In code cell 1, he imports two lines of code. Line 1 is: import requests. Line 2 is: import webbrowser.* [Video description ends]

Following that, we define the URL to which we will make our first GetRequest in this demo, and this is to wikipedia.org.

[Video description begins] *He imports the following code in code cell 2: url = "https://www.wikipedia.org/".* [Video description ends]

Next, we will go ahead and make our GetRequest. So first, we will just pass along the URL like we did in the previous videos, and will not specify any parameters.

[Video description begins] *In code cell 3, he imports the following code: resp_obj = request.get(url).* [Video description ends]

This returns to us a response object, as we would expect, so we can just go ahead and run the cell, and then, we will explore the response object by first accessing the URL field.

[Video description begins] *He imports the following code in cell 4: print(resp_obj.url).* [Video description ends]

So this confirms to us that the URL to which the request was sent is wikipedia.org, and we will now see how we can also make use of the web browser instance in order to access this URL.

[Video description begins] *The following output appears below code cell 4: https://www.wikipedia.org/.* [Video description ends]

So we call the function webbrowser.open, and the argument to this is our wikipedia.org URL.

[Video description begins] *In code cell 5, he imports the following code: webbrowser.open(resp_obj.url).* [Video description ends]

If the URL is valid, the value True will appear in the notebook, and now, this will bring up the Wikipedia website in a new browser window.

[Video description begins] *The following output appears below code cell 5: True* [Video description ends]

[Video description begins] *A window titled Wikipedia opens. The URL is https://www.wikipedia.org. The logo of Wikipedia is displayed. Below it, there is a Search bar.* [Video description ends]

So we now have an idea of how we can make use of this web browser instance.

[Video description begins] *He scrolls down the page. The page contains the following options: Commons, Wikivoyage, Wikibooks, etc.* [Video description ends]

We will now proceed with our demo, for which we return to the jupyter Notebook, and now, we will first ask the user for a search term, and this is a search which will be performed on the YouTube site.

[Video description begins] *He shifts back to the jupyter notebook window.* [Video description ends]

[Video description begins] *He imports the following code in code cell 6: search_term = input("Enter the term you need to search: ").* [Video description ends]

So here, we will accept an input from the user, and then, when we run the cell, we will be prompted for the input, and the specific term which I'm going to search for on YouTube will be skillsoft.

[Video description begins] *An input field for Enter the term you need to search appears as an output. In the input field, he types skillsoft.* [Video description ends]

Once this has been accepted, we will set another URL variable, and this, I'm going to set to youtube.com/search.

[Video description begins] *In code cell 7, he imports the following code: URL = 'https://www.youtube.com/search'.* [Video description ends]

So this will be the URL endpoint for our next GetRequest. But on this occasion, we will be passing along not just the URL, but also a dictionary of parameters to the GetRequest.

[Video description begins] *He imports the following code in code cell 8: PARAMS = {'q' : search_term}.* [Video description ends]

So here, we specify a dictionary which includes just a single key and value. However, depending on the API which has been implemented, there can be multiple parameters which are passed along with your GetRequest. Since we're making a call to youtube.com/search, we pass along the search term as the value of the parameter q. So now that we have our URL and parameters dictionary, we can invoke requests.get once more in order to make our GetRequest, and on this occasion, we pass along both the URL and the params dictionary.

[Video description begins] *He imports the following code in code cell 9: r_get = request.get(url = URL, params = PARAMS).* [Video description ends]

All right, so when we run the cell, we do get a response back, and we can confirm that this operation was a success by looking at the status code of the response, and yes, this returns a value of 200, confirming that this was a successful operation.

[Video description begins] *In code cell 10, he imports the following code: print(r_get.status_code).* [Video description ends]

[Video description begins] *The following value appears below the code cell: 200.* [Video description ends]

Now, when we print out the URL of the response object, what we see is not the URL which was passed along to our GetRequest, but something a little different. A request was made to youtube.com/search, and a response points to the URL youtube.com/results along with the parameter of skillsoft which we had passed as the search term.

[Video description begins] *He imports the following code in code cell 11: print(r_get.url). Below the code cell, an output appears. It displays the following URL: https://www.youtube.com/results?q=skillsoft. The following code is displayed on the screen: URL ='https://www.youtube.com/search'.* [Video description ends]

So while our request to wikipedia.org returned a response object which pointed to the same page, on this occasion we make a request to one URL, and a response points to another. So in order to view the response, we can either explore the JSON contents directly in our notebook, or alternatively, a better option will be to view the contents in a web browser, and for that, we make use of the web browser module once more, and call webbrowser.open and pass along the URL of our response object.

[Video description begins] *The code displayed on the screen is now removed.* [Video description ends]

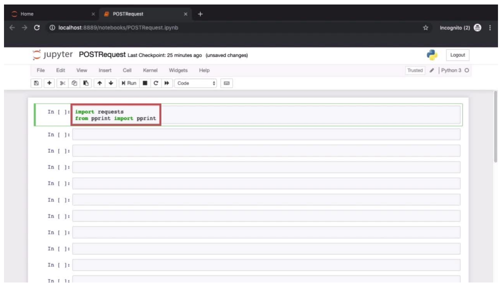
[Video description begins] *In code cell 11, he imports the following code: webbrowser.open(r_get.url).* [Video description ends]

So this returns a value of True in the notebook, and a YouTube page should pop up on your web browser, and the results for the search term of skillsoft will be available.

[Video description begins] *The following value appears below the code cell: True. A skillsoft - YouTube page opens in a browser window. The Search bar contains the text: skillsoft. Below it, the following two results are displayed: Skillsoft Aspire Learning Journeys Demo and Recognizing Gender Bias and Barriers.* [Video description ends]

So we have effectively simulated what most users of YouTube will have performed, where we go into youtube.com, perform a search, and we are returned a number of results, and while we can explore these results in a web browser, it is also possible for us to access the results using the request object in a Python notebook, and then pass what has been returned.

6. Video: A Basic POST Request (it_pywhpydj_01_enus_06)



```
import requests
from pprint import pprint
```

Objectives

- *use the Requests package to construct a POST request that includes a set of key-value pairs to be submitted to a server*

[Video description begins] *Topic title: A Basic POST Request. Your host for this session is Kishan Iyer.* [Video description ends]

So far in this course, we have covered the use of get requests using the Python requests package. Get request are typically used when we wish to get information from a particular server.

[Video description begins] *A jupyter notebook titled POSTRequest is open in a browser window. It has a menu bar with the following options: File, Edit, View, Insert, etc. There is a tool bar with the following icons: Save, Add, Cut, Run, etc. In the center pane, a few code cells are present.* [Video description ends]

Now, what if we wish to hand over information or give information to a server instead? Well, typically then, a POST request is made, where we will be posting data to that server, and this is an operation which is once again possible using the Python requests package. In this demo, we will see how we can construct a POST request, and we'll send over a POST request to a URL, which is specifically meant for the purpose of testing out your HTTP requests.

[Video description begins] *He imports two lines of code in code cell 1. Line 1 is: import requests. Line 2 is: from pprint import pprint.* [Video description ends]

We begin writing our code, and once again we import the request and pretty print modules, which we will require, and one thing to keep in mind before we proceed any further is that any website will not allow you to just make POST requests for free without being authenticated.

This is very much understandable from a security standpoint because we will be posting information, and any server will need to be very careful about whom they allow to make such requests. For that reason, we will be effectively simulating or testing POST requests in this demo. To start off, we will make a post request by calling the function requests.post, and the arguments to this include the URL to which the POST request is being made, along with the data contained within the request.

[Video description begins] *The following code appears in code cell 2: r_post = request.post('https://en.wikipedia.org/w/index.php', data = {'search' : 'Skillsoft'}).* [Video description ends]

So this is a request to wikipedia.org/w/index.php, and our data is in the JSON format. It includes a key called search and a value called Skillsoft. Now keep in mind that a POST request is made when we wish to send over data to a particular server, and this data can include anything, and it is up to the server to decide how it is used. In this case, that data will be used to perform a search on the Wikipedia website. So when we make this POST request, we capture the response, which is returned in this variable, r_post, and then we can view the status_code to confirm that this was a successful operation, and the code of 200 does confirm that is the case.

[Video description begins] *The following code appears in code cell 3: r_post.status_code.* [Video description ends]

[Video description begins] *An output appears below code cell 3. It reads: 200.* [Video description ends]

We can take a look at the type of this r_post variable, and just like with the response of a get request, this is also of type requests.models.Response.

[Video description begins] *In code cell 4, the following code appears: type(r_post).* [Video description ends]

[Video description begins] *An output appears below code cell 4. It reads: request.model.Response.* [Video description ends]

All right, we now move ahead and take a look at the contents of the response body, and with that we access the text field of this response object.

[Video description begins] *The following code appears in code cell 5: pprint(r_post.text).* [Video description ends]

We use pretty print to print out the data, and what we observe is that what is returned is in fact an HTML file.

[Video description begins] *An HTML file appears below code cell 5 as an output.* [Video description ends]

Now, before we view the HTML file and render it in a browser, what we will do is to save the contents in a file on our own file system.

[Video description begins] *The following code appears in code cell 6: with open('skillsoft.html', 'wb') as f: for chunk in r_post.iter_content(chunk_size=10000): f.write(chunk).* [Video description ends]

So I'm going to make use of the Python with statement in order to save the contents of this HTML file in a file called skillsoft.html, and note that I'm first creating such a file and opening it using the open function, and inside the width block, we are iterating through the contents of our response body in the form of chunks, and then writing each chunk to this skillsoft.html file. This is saved down the HTML, which we saw previously in this HTML file, and then following that, we will open up that file using the Google Chrome browser.

[Video description begins] *The following code appears in code cell 7: !open -a "Google Chrome" skillsoft.html.* [Video description ends]

For that I'm executing the open command from the shell, and again, to run a shell command from within a jupyter notebook, you need to prepend it with an exclamation point. The arguments to the open command include the application which will be used to open up a file, and this is followed by the -a flag, and the app, of course, is Google Chrome, and then at the end, we specify the file which needs to be opened, which in our case is skillsoft.html. Do note that the skillsoft.html file has been saved in the same project workspace directory where we have created our jupyter notebooks. All right, when we run this open command, this will open up a new Google Chrome instance, and you can see the contents of the HTML file rendered here.

[Video description begins] *A page titled Skillsoft - Wikipedia opens in a browser window. The URL is: /Users/loonycorn/Desktop/Requests/skillsoft.html. A section titled Skillsoft is present here. It has the following information: Type, Industry, Founded, Products, etc.* [Video description ends]

Clearly a lot of the formatting information is not available in this file, probably because this Wikipedia page references some CSS files, which I clearly don't have on my local file system. However, all of the main information for the page is available in this HTML.

[Video description begins] *He scrolls down the page. It has the following sections: History, References, Navigation menu, Print/export, Languages, etc.* [Video description ends]

So with that, we have successfully made our first POST request. In this particular instance, the JSON data which we supplied with our POST request was used in order to perform a search on the Wikipedia website. In other cases, however, that data may be used for different purposes, maybe to be stored on the server itself, and a common use case for POST requests can be on any website, where you can post a review.

All of the review data along with the review text, the headers, a star rating if there is one, will be included within the POST request body. A POST request can also include a file, and to see how that can be done, we return to our jupyter notebook, and we will first create a file which we can include within our POST request.

[Video description begins] *He shifts back to the jupyter notebook page. The following code appears in code cell 8: !echo "This is a text file that will be attached to a POST Request" > test.txt.* [Video description ends]

To do that, I'm going to create a file called test.txt and this will contain the text. This is a text file that will be attached to a POST request. Note the use of the echo command, which is prepended by an exclamation point, and the redirection of this echo output to the test.txt file. We follow this up by examining the contents of the test.txt file just to make sure that our text is indeed available within that, and for that we make use of the cat command, and we can confirm that we now have a text file which contains one sentence.

[Video description begins] *The following code appears in code cell 9: !cat test.txt*. [Video description ends]

[Video description begins] *An output appears below code cell 9. It reads: This is a text file that will be attached to a POST Request*. [Video description ends]

All right, we move along, and then form the URL to which we will be making our second POST request, which includes this file.

[Video description begins] *In code cell 10, the following code appears: url = 'http://httpbin.org/post'*. [Video description ends]

This is a website called httpbin.org, and this is specifically meant for the purposes of testing out your requests. Though it won't be possible for us to go to the website and see the effects of our post on that site itself, we will at least get a response object back which will confirm whether our operation was a success and whether our POST request was formed correctly. Moving along to what else will be attached to our POST request, this will include the file which we just created, and note that we are creating a Python dictionary here, which includes a key called files, and the corresponding value is the binary of the test.txt file which we created, and this is available using the open function.

[Video description begins] *Two lines of code appear in code cell 11*. [Video description ends]

[Video description begins] *He highlights code line 1 in code cell 11. It reads: files = {'files': open('test.txt', 'rb')}*. [Video description ends]

We then move along to the values or the data which will be included in our POST request, and this is included in a dictionary called values.

[Video description begins] *He highlights code line 2 in code cell 11. It reads: 'values= {'upload_file' : 'test.txt' , 'OUT':'csv'}*. [Video description ends]

Here we include the name of our uploaded file, which is test.txt, and, we also include a field called OUT, which has been set to the value csv. Note that the exact contents of this values dictionary is not really important. What our POST request call to httpbin.org will confirm is whether the file and the data which is being attached to the POST request has been formatted correctly. So now we can go ahead and take a look at the files variable which we have created, and this confirm that this includes an io_BufferedReader that is the binary version of our test.txt file.

[Video description begins] *The following code appears in code cell 12: files* [Video description ends]

[Video description begins] *An output appears below code cell 12. It reads: {'files': <_io.BufferedReader name='test.txt'>}*. [Video description ends]

Moving along now to making our POST request.

[Video description begins] *The following code appears in code cell 13: r_post = requests.post(url,files=files,data=values)*. [Video description ends]

So we call requests.post once more, and here we specify three different arguments. The URL to which we are making the POST request, the files which we are attaching along with our POST request, and the data itself in JSON format. Now once again, it is up to the server to decide what it does with the data which it receives in the POST request. This includes the files and also the JSON data. However, httpbin.org will only perform a validation of our data. When we run this, no error is thrown, and we can confirm the status code of the response object, which gives us a value of 200, and this confirms to us all that we need to know, that the way in which we constructed our POST request is correct.

[Video description begins] *In code cell 14, the following code appears: print(r_post.status_code). The following value appears below the code cell: 200*. [Video description ends]

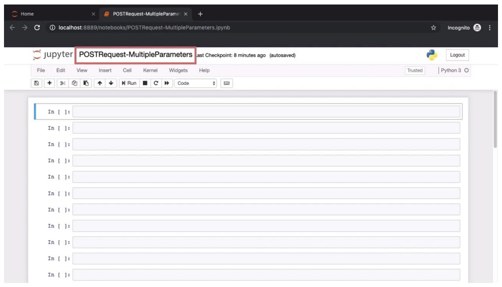
We can take a look at the URL for this POST request, and this is what we had passed along to the POST function.

[Video description begins] *The following code appears in code cell 15: print (r_post.url). An output appears below the code cell. It reads: http://httpbin.org/post* [Video description ends]

Examining the full response body, we can see all of the data which was included in our request, including the contents of our text file.

[Video description begins] *In code cell 16, the following code appears: pprint(r_post.text). An output appears below the code cell. It contains a set of code lines*. [Video description ends]

7. Video: A POST Request with Multiple Parameters (it_pywhpydj_01_enus_07)


jupyter
POSTRequest-MultipleParameters
Last Checkpoint: 8 minutes ago (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

Objectives

- *submit data to pastebin.com using their APIs and with a POST request*

[Video description begins] *Topic title: A POST Request with Multiple Parameters. Your host for this session is Kishan Iyer.* [Video description ends]

I had earlier mentioned that websites and servers typically don't allow POST request to be made, unless you register with them first, and they can keep track of who is making those requests.

[Video description begins] *A jupyter notebook titled POSTRequest-MultipleParameters is open in a browser window. It has a menu bar with the following options: File, Edit, View, Insert, etc. There is a tool bar with the following icons: Save, Add, Cut, Run, etc. In the center pane, a few code cells are present.* [Video description ends]

However, if you would like to see the effects of your POST request on a website, you can create an account on pastebin.com. This is a service which allows you to store any form of text online, and then share it.

[Video description begins] *A banner appears. It displays the following URL: https://pastebin.com.* [Video description ends]

This is often used by programmers, in order to share their code with the general public. But the feature we are interested in for this demo is the fact that it allows us to post data to the website using a POST request. To do that, however, you will not only need to register on the website, but you will also need to get your API key. Once that is done, we can begin coding, and as usual, we will start off by importing the modules which we require.

[Video description begins] *He imports three lines of code in code cell 1. Line 1 is: import requests. Line 2 is: from pprint import pprint. Line 3 is: import webbrowser.* [Video description ends]

We then create the URL which we will be using in order to make our post, and to do this, we will be invoking the Pastebin API. Following this, we specify the data which we will be submitting in a POST request.

[Video description begins] *The following code appears in code cell 2: post_link = "http://pastebin.com/api/api_post.php".* [Video description ends]

[Video description begins] *The following code appears in code cell 3: payload = "{'username': 'john', 'email': 'john@john.com'}".* [Video description ends]

This will make up our payload, and you can see that this is in the JSON format. You can include any data which you wish over here. It just needs to be some text, which will be posted on that website, and I'm including the username and email of a fictitious person named John, and following this, another detail which you will need to include in a POST request will be an API_KEY, and this is something which I have captured from my Pastebin account.

[Video description begins] *The following code appears in code cell 4: API_KEY = ' 068ef8c5332427952bd43af129886ca0 '.* [Video description ends]

So once we have all of these details, we will create a dictionary called parameters. This is going to include all of the data which we will include in a POST request, and the details to be included in such a request can be found in the Pastebin API documentation, at this URL.

[Video description begins] *A code appears in code cell 5. It reads: parameters = {'api_dev_key':API_KEY,'api_option':'paste','api_paste_code':payload, 'api_paste_format':'python'}.* [Video description ends]

[Video description begins] *A banner appears. It displays the following URL for Pastebin API documentation: https://pastebin.com/api.* [Video description ends]

So the POST request body needs to include a field called api_dev_key, which of course includes the API_KEY corresponding to your account, a value of paste for the api_option says that you want to create a new paste, and then the actual contents of the data you're pasting is included in the field api_paste_code. We follow this up by including a field called api_paste_format, and this will determine the syntax highlighting which will be available on the Pastebin website for the data which you have posted.

So I'm just going to use the value of python for this. With that done, all of the data which we require in order to make up POST request to Pastebin has been set, and we can just make a POST request by again calling requests.post. The arguments which we passed to this include the URL for the POST request, along with our parameters dictionary, and when we run this cell, a good sign is that there is no error.

[Video description begins] *The following code appears in code cell 6: r_post= requests.post(post_link, data=parameters).* [Video description ends]

But now, we will go ahead and create an if and else block, where we will check the status code which was returned in the response object, and then if this is equal to 200, we print out the fact that this request was successful, and in order to view the data which has been posted, the response object will include a URL in the response body, and this is accessible from the field r_post.text.

[Video description begins] *Five lines of code appear in code cell 7. Line 1 is: if (r_post.status_code == 200):. Line 2 is: print("The request to the url was successful."). Line 3 is: print ("You can find the code pasted on this link: {}".format(r_post.text)). Code line 4 reads: else:. Code line 5 is: print("The request was not successful.").* [Video description ends]

And now when we run this, we get the link to view the data which we have posted to Pastebin. You can either click on that link, or alternatively, you can use the webbrowser.open function in order to bring it up in a new browser instance, and with that done, you can see that our JSON data is available in this field.

[Video description begins] *Two lines of output appears below code cell 7. Line 1 is: The request to the url was successful. Line 2 is: You can find the code pasted on this link: https://pastebin.com/HkfLMMjf.* [Video description ends]

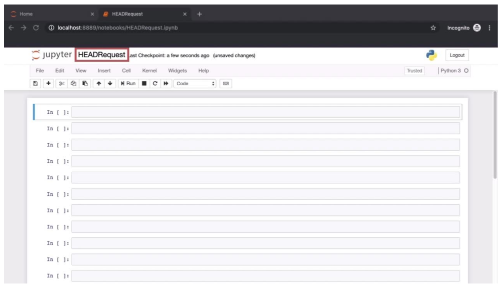
[Video description begins] *The following code appears in code cell 8: webbrowser.open(r_post.text). The following value appears below code cell 8: True.* [Video description ends]

It is available in both a Python-formatted field, and also the RAW paste data is available, and so we have now successfully made use of a POST request in order to post data to a website.

[Video description begins] *A window titled PASTEBIN opens up. The Menu bar has a + new paste button and the following options: API, tools, faq, and deals. A Search bar is also present here. A page titled Untitled is open in the center pane. There is an Editor area with the following buttons: raw, download, clone, embed, etc. It contains a line of code, which reads: {'username': 'john', 'email': 'john@john.com'}. A section titled RAW Paste Data is present below the Editor area. It contains the same code line.* [Video description ends]

While Pastebin allows us to view the data which has been posted, it can be used for several other purposes. Maybe the server can be coded up, in order to perform some kind of processing of the data before being published, or it could just store the data which has been posted, and then use it for analysis. For the purposes of this course, however, what is important is how exactly we can make such a POST request, and this is what we have covered.

8. Video: The HEAD Request (it_pywhpydj_01_enus_08)


Home
HEADRequest
localhost:8889/notebooks/HEADRequest.ipynb
Incognito
jupyter HEADRequest Last Checkpoint: a few seconds ago (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code
In [ ]:

Objectives

- *retrieve metadata for a resource by means of a HEAD request*

[Video description begins] *Topic title: The HEAD Request. Your host for this session is Kishan Iyer.* [Video description ends]

The types of request which we have covered so far in this course have included GETRequests, which are often used in order to get data from a server.

[Video description begins] *A jupyter notebook titled HEADRequest is open in a browser window. It has a menu bar with the following options: File, Edit, View, Insert, etc. There is a tool bar with the following icons: Save, Add, Cut, Run, etc. In the center pane, a few code cells are present.* [Video description ends]

We have also covered POSTRequests and this can be used in order to post data or to submit information to a server. We will now explore one more special kind of request and this is known as a HEADRequest. Now in many ways, a HEADRequest is similar to a GETRequest except that HEADRequests don't return any content. It is often used in order to check the information of the resource before downloading it using get.

For example, if you want to confirm the existence of a large file before you in fact kick off a very long download process, you can perform this check using the HEADRequest which will return among other details the size or the length of the file. So let us go ahead and invoke such a HEADRequest.

[Video description begins] *He imports three lines of code in code cell 1. Line 1 is: import requests. Line 2 is: from pprint import pprint. Line 3 is: import json.* [Video description ends]

So we begin as usual, by importing all of the modules which we require and we will go ahead and make a HEADRequest by calling requests.head.

[Video description begins] *The following code appears in code cell 2: r_head = requests.head ('http://example.com').* [Video description ends]

The specific URL we will use this time is example.com and this will return a response object. As always, we will confirm that this request was a success by checking the status code in the response, and yes, this returns the value of 200 confirming that our request was successful.

[Video description begins] *The following code appears in code cell 3: r_head.status_code.* [Video description ends]

[Video description begins] *An output appears below the code cell. It reads: 200.* [Video description ends]

We now move along to one of the other fields in our response object and we access the body of the response using the text field and this returns nothing.

[Video description begins] *The following code appears in code cell 4: r_head.text. Below it, the output reads:* ' '. [Video description ends]

So as I had mentioned, a HEADRequest is rather similar to a GETRequest, except that the HEADRequest does not include any content. In fact, if we try to access the content by using the content field of our response object, we run the cell and this returns, again, a blank string.

[Video description begins] *The following code appears in code cell 5: r_head.content. An output appears below the code cell. It reads: b* ' '. [Video description ends]

The lowercase b before the blank string denotes the fact that the content is represented as bytes. So where exactly is the data which will be useful for us when we make a HEADRequest? As touched upon, this is often used to check for the existence of a resource and to access its metadata rather than access the data itself, and all of the data which is of interest to us is available in the headers of the response.

[Video description begins] *The following code appears in code cell 6: pprint(r_head.headers).* [Video description ends]

So we now go ahead and use the pprint function in order to print out the headers, and all of the information which is useful when invoking a HEADRequest is right here.

[Video description begins] *A set of content appears below the code cell. It includes: 'Content-Encoding', 'Content-Type', 'Content-Length', etc.* [Video description ends]

So all of the metadata for the actual information which will be returned by a GETRequest to example.com is right here. For example, the last field here which represents the content length shows that the size of the data will be 606 bytes, and the content type field confirms that the data is in the form of HTML.

[Video description begins] *In the output, he highlights the following text: 'Content-Length': '606'.* [Video description ends]

[Video description begins] *He highlights the following text: 'Content-Type': 'text/html; charset=UTF-8'.* [Video description ends]

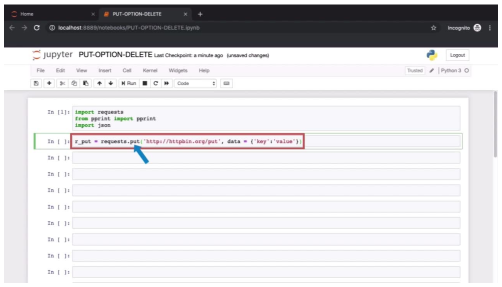
In fact, since this data is in json form, we can access each of these individual fields. So first, we access the content length field in the headers and this returns the value of 606, and then we access the content-type field as well, which returns text/html.

[Video description begins] *The following code appears in code cell 7: pprint(r_head.headers['content-length']). An output appears below code cell 7. It reads: '606'.* [Video description ends]

[Video description begins] *The following code appears in code cell 8: pprint(r_head.headers['content-type']). An output appears below the code cell. It reads: 'text/html; charset=UTF-8'.* [Video description ends]

So why exactly is this information useful? Well, consider that you have built some kind of web crawler, and you're crawling through all of the hyperlinks on a particular website. You may first wish to find out what data is accessible from the link before you potentially download it. For example, you may only be interested in image files which are less than five megabytes in size. In such a case, you can take a look at the content length to figure out what the size of the file is and also the content type to confirm that it is an image. This will save you the trouble of potentially having to download a very large file and then discarding it because it does not contain the information which you want.

9. Video: The PUT, OPTIONS, and DELETE Requests (it_pywhpydj_01_enus_09)


Home
PUT-OPTION-DELETE
localhost:8889/notebooks/PUT-OPTION-DELETE.ipynb
Incognito
jupyter PUT-OPTION-DELETE Last Checkpoint: a minute ago (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

```
In [1]: import requests
        from pprint import pprint
        import json

In [ ]: r_put = requests.put('http://httpbin.org/put', data = {'key':'value'})

In [ ]:
In [ ]:
In [ ]:
In [ ]:
In [ ]:
In [ ]:
In [ ]:
In [ ]:
```

Objectives

- *invoke PUT, OPTIONS, and DELETE requests and recognize the specific use case for each of them*

[Video description begins] *Topic title: The PUT, OPTIONS, and DELETE Requests. Your host for this session is Kishan Iyer.* [Video description ends]

We will now continue exploring some of the other kinds of request which can be made by making use of the request package in Python.

[Video description begins] *A jupyter notebook titled PUT-OPTION-DELETE opens. It has a menu bar with the following options: File, Edit, View, etc. There is a tool bar with the following options: Save, Add, Cut, etc. Beneath it, a set of code cells is present.* [Video description ends]

We have already covered GET, POST, and HEAD requests, and in this new jupyter notebook, we will be exploring the PUT, OPTION, and DELETE requests. We begin by importing the modules which we require, which includes requests, pprint, and json, and then the first request we make is a PUT request.

[Video description begins] *Three lines of code appear in code cell 1. Code line 1 is: import requests. Code line 2 is: from pprint import pprint. Code line 3 is: import json.* [Video description ends]

[Video description begins] *A code line appears in code cell 2. It reads: r_put = requests.put ('http://httpbin.org/put', data = { 'key' : 'value' }).* [Video description ends]

So what exactly does the PUT request do? Well, it is in many ways similar to a POST request, with one significant difference. A PUT request is supposed to be idempotent. That is, it'll have the same effect whether you make the request once or multiple times. It is often used in order to overwrite an existing resource rather than just modify it. The full difference between PUT and POST request can be found at the URL which you see on the screen.

[Video description begins] *A banner appears. It contains the following URL: https://restfulapi.net/rest-put-vs-post/.* [Video description ends]

So to make such a PUT request, we call requests.put, and once again we will be making use of httpbin.org in order to test out our PUT request.

[Video description begins] *He highlights the following part of code in code cell 2: requests.put.* [Video description ends]

[Video description begins] *He highlights the following part of code in code cell 2: ('http://httpbin.org/put',.* [Video description ends]

So we need to attach some data to this request, just as we did with POST requests, and this has to be in the JSON format.

[Video description begins] *He highlights the following part of code in code cell 2: data = { 'key' : 'value' }).* [Video description ends]

So the data which is usually included in such a PUT request will include the data which needs to be written at the server, and this will be in the form of key and value pairs. So we go ahead and run the cell, and our PUT request has been made.

[Video description begins] *A code line appears in code cell 3. It reads: r_put.status_code.* [Video description ends]

We can confirm that this went through successfully by examining the status code, and yes, this returns a value of 200, and we can confirm that what was returned from our PUT request was indeed a response object.

[Video description begins] *An output appears for code line 3. It reads: 200.* [Video description ends]

So we check the type, and this is indeed a response object.

[Video description begins] *A code line appears in code cell 4. It reads: type(r_put). An output appears for code line 4. It reads: requests.models.Response.* [Video description ends]

Now, what exactly is the data which has been returned in this response object? For that, we examine the text field, and in the case of httpbin, you will see a response like this.

[Video description begins] *A code line appears in code cell 5. It reads: r_put.text.* [Video description ends]

[Video description begins] *An output appears for code line 5. It contains the following information: "Content-Type", "Accept-Encoding", etc.* [Video description ends]

As I had mentioned, PUT requests are often used in order to set the value at the server, and this needs to be an idempotent operation. This is a reason why PUT requests are often used in order to perform setter operations, where you set the value of different fields at the server. This could be to set the values of different configuration properties, or even to write data to a database. But this, of course, depends on the specific implementation. We now move along to the next type of request, which is the OPTIONS request.

[Video description begins] *A code line appears in code cell 6. It reads: r_option = requests.options ('https://httpbin.org/get').* [Video description ends]

Now, this is one which is used in order to describe the communication options for the target resource. We will see exactly what that means by first invoking such a request, for which we call the function requests.options. Once again, we make use of the httpbin website. So here we are making a call to its get API, and then we capture the returned value and examine its type.

[Video description begins] *A code line appears in code cell 7. It reads: type(r_option).* [Video description ends]

Once again, this is a response object.

[Video description begins] *An output appears for code line 7. It reads: requests.models.Response.* [Video description ends]

So if you go ahead and examine the contents in the body of this object, we see that it is a blank string, and in this manner, it is similar to a HEAD request, in that the point of making an OPTIONS request is not to capture the response body, but just to examine some of the metadata for the resource, and where exactly is this metadata available?

[Video description begins] *A code line appears in code cell 8. It reads: r_option.text.* [Video description ends]

[Video description begins] *An output appears for code line 8. It reads:* ' '. [Video description ends]

This is, of course, in the headers, just as we saw with the HEAD request.

[Video description begins] *A code line appears in code cell 9. It reads: pprint(r_option.headers).* [Video description ends]

So we will now go ahead and print out these headers, and out of these, there are a handful of fields which are of interest to us when we make such an OPTIONS request.

[Video description begins] *An output appears for code line 9. It contains the following information: Access-Control-Allow-Credentials, Access-Control-Allow-Methods, Access-Control-Allow-Origin, Access-Control-Max-Age, etc.* [Video description ends]

One of these is the Allow field, and this is something which we saw previously, and this gives us the types of request operations which can be performed on the resource.

[Video description begins] *In the output for code line 9, he highlights the following text: 'Allow' : 'HEAD, GET, OPTIONS',.* [Video description ends]

In this particular case, the allowed operations are OPTIONS, GET, and HEAD requests. Now, this is a field which we saw previously in the headers when we made a GET request.

[Video description begins] *He highlights the entire output for code line 9.* [Video description ends]

But what we did not see were a couple of fields which are visible here, and these are the fields related to CORS, which is short for Cross-Origin Resource Sharing, and is a mechanism which uses HTTP headers in order to define how a resource can be shared from other servers. So the fields which are specific to CORS here include Access-Control-Allow-Credentials, Access-Control-Allow-Methods, Access-Control-Allow-Origin, and Access-Control-Max-Age.

[Video description begins] *He points to four fields in the output for code line 9.* [Video description ends]

I won't get into the specific details of CORS here, but it is enough for you to know that these four different fields are useful when using CORS. We now move along to the next type of request, which can be done using the Request package, and this is a DELETE request.

[Video description begins] *A code line appears in code cell 10. It reads: r_delete= requests.delete ('http://httpbin.org/delete').* [Video description ends]

As the name implies, this is something which is used if you wish to delete a specific resource at the server. So here you need to specify the URL endpoint for the resource as the argument, and once again, we'll be making use of httpbin in order to test out our DELETE call.

So here we are simulating the deletion of a resource at the specified URL rather than actually deleting something on a server. So we make this call to requests.delete, and then we take a look at the status code for our operation, and the value of 200 confirms that this operation was a success.

[Video description begins] *A code line appears in code cell 11. It reads: r_delete.status_code.* [Video description ends]

[Video description begins] *An output appears for code line 11. It reads: 200.* [Video description ends]

We check the type of object which was returned, which, as usual, is a response object, and then we capture the full response body and then print it out by accessing the text field of our response object.

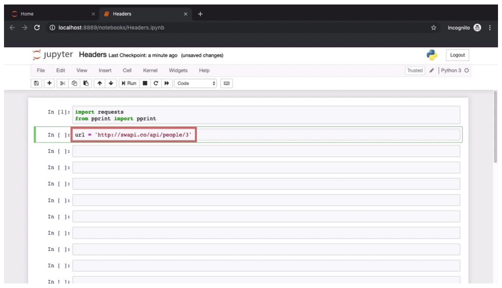
[Video description begins] *A code line appears in code cell 12. It reads: type(r_delete).* [Video description ends]

[Video description begins] *An output appears for code line 12. It reads: requests.models.Response.* [Video description ends]

[Video description begins] *A code line appears in code cell 13. It reads: r_delete.text. An output appears for code line 13. It contains the following information: Accept-Encoding, User-Agent, etc.* [Video description ends]

So while this is the data which you see returned by httpbin, depending on the specific implementation, you may see a different message on different servers. So we have now seen three other forms of HTTP requests which can be made using the Python Requests package. With that, we have now covered the different types of request which can be made, and in the remaining videos of this course, we will take a look at some of the configurations which allow us to customize the types of requests which we make to servers.

10. Video: Working with Request and Response Headers (it_pywhpydj_01_enus_10)



```
In [1]: import requests
        from pprint import pprint

In [ ]: url = 'http://swapi.co/api/people/3'
```

Objectives

- *set the headers in an HTTP Request and parse the headers in a Response object*

[Video description begins] *Topic title: Working with Request and Response Headers. Your host for this session is Kishan Iyer.* [Video description ends]

One of the main reasons for using the request package in Python is so that developers don't need to construct an HTTP request by themselves.

[Video description begins] *A jupyter notebook titled Headers opens. It has a menu bar with the following options: File, Edit, View, etc. There is a tool bar with the following options: Save, Add, Cut, etc. Beneath it, a set of code cells is present.* [Video description ends]

A lot of the underlying details for HTTP requests, including the setting of headers, is managed by the package itself. While it is helpful to have such details abstracted from the developer, it is also necessary to allow them to edit the headers and other details when required. In this demo, we will take a look at how we can set specific fields in the header when making an HTTP request. So we begin by importing the requests and the pprint modules, and then we follow that up by specifying the URL to which we will be making a GET request.

[Video description begins] *Two lines of code appear in code cell 1. Code line 1 is: import requests. Code line 2 is: from pprint import pprint.* [Video description ends]

[Video description begins] *A code line appears in code cell 2. It reads: url = 'http://swapi.co/api/people/3'.* [Video description ends]

We will again make use of the Star Wars API. Now all of the GET request which we have made in this course so far have included a URL, and on some occasions, we have also passed along parameters to the GET request. On this occasion though, we will also set one of the fields in the headers in our request.

[Video description begins] *A code line appears in code cell 3. It reads: headers = { 'user-agent' : 'Googlechrome'}.* [Video description ends]

To do this, we create a dictionary, and here, we specify that the value in our header for the field user-agent will be Googlechrome. The user-agent field is typically used by an application to indicate which app the request is arising from. The server may use that information in order to customize its response depending on the app from which you are invoking the request, whether it is a web browser or an app on an Android device.

If that is the case, it is often beneficial for the one making the request to specify which app the request is originating from, and that is when you specify the user-agent, as we have done here. We will then proceed to make this GET request, where we pass along the URL and also the header, which we have created as the value of the headers field.

[Video description begins] *A code line appears in code cell 4. It reads: resp = requests.get (url, headers=headers).* [Video description ends]

So when we run the cell, we collect a response and then we can take a look at the headers of the response object.

[Video description begins] *A code line appears in code cell 5. It reads: resp.headers.* [Video description ends]

So we run this, and what we see is a standard response object, as we have seen already in this course.

[Video description begins] *An output appears for code line 5. It contains the following information: 'Content-Type', 'X-Frame-Options', etc.* [Video description ends]

There is no indication of the header which we included as part of the response object over here. But we have now seen how we can invoke a GET operation by specifying the headers of our request object. Now in order to access the headers in our response, we can treat it as we would a Python dictionary, and here, we access the value of the content-type header field, and for a specific response, this is of type application/json.

[Video description begins] *A code line appears in code cell 6. It reads: resp.headers ['content-type'].* [Video description ends]

[Video description begins] *An output appears for code line 6. It reads: ' application/json'.* [Video description ends]

We can now go ahead and make one more GET request.

[Video description begins] *A code line appears in code cell 7. It reads: resp_obj = requests.get ('https://en.wikipedia.org/wiki/Monty_Python').* [Video description ends]

So in this case, we will access the article from Monty_Python on Wikipedia, and then we will capture the response and take a look at the headers.

[Video description begins] *A code line appears in code cell 8. It reads: resp_obj.headers.* [Video description ends]

On this occasion, there are many more fields in the headers, which are returned in the response, and this confirms to us that even the data within the headers of the response is something which is very much configurable by the application whose server endpoint we are hitting.

[Video description begins] *An output appears for code line 8. It contains the following information: 'Content-Type', 'X-Content-Type-Options', etc.* [Video description ends]

At this point though, what if we wish to access our original request headers, so we wish to see exactly what was set by default when we made a GET request, and one way to do that is to access the request object through the response object.

[Video description begins] *A code line appears in code cell 9. It reads: resp_obj.request.headers.* [Video description ends]

The request object is available through resp_obj.request, and if we wish to access the headers in a request object, we can access its headers field. So here, we run the cell, and we can see exactly what was included by default when we invoked a request.

[Video description begins] *An output appears for code line 9. It contains the following information: 'Accept-Encoding', 'Connection', etc.* [Video description ends]

For example, you can see here that the User-Agent field was set to python-request, and the version of a request package was also included here.

[Video description begins] *He highlights a section of the output for code line 9. It reads: { 'User-Agent' : 'python-requests/2.22.0'.* [Video description ends]

So when we invoked a GET request, the Wikipedia site knew that this request was coming in from an application which was making use of the Python HTTP requests package, and even the specific version of that package was available. This, however, can prove to be a problem in some instances. Consider that you're running some kind of test and you wish to see what a webpage might look like on the Internet Explorer browser, for example. Well, fortunately, we have seen that it is possible for us to explicitly set the value of the different fields in our header object, and on this occasion, we can make an HTTP request, and here we specify the value of the User-agent field in our header to be Internet Explorer.

[Video description begins] *A code line appears in code cell 10. It reads: r = requests.get ('http://httpbin.org/user-agent', headers = { 'User-agent' : 'Internet Explorer/2.0' }).* [Video description ends]

So by adjusting the value of the User-agent field in the header, you can simulate GET requests from different web browsers, and then programmatically examine the contents of a page as they would appear on those web browsers. So we can now run this cell and then capture the JSON version of the response, and to do that, we can call the response object's json function.

[Video description begins] *A code line appears in code cell 11. It reads: data = r.json ().* [Video description ends]

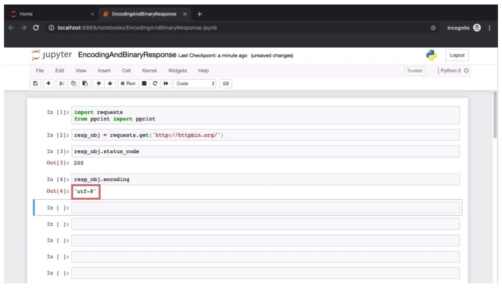
This is one more option which is available to us in order to pass the response object. But an important factor to consider is that this json function will function without throwing errors only if the response does contain JSON data. As we have seen, you can check for the type of data which is returned within the response object by examining the content-type field in the response header, and in the case of this response from httpbin.org/user-agent, we will have a user-agent field in the response, which we can access as we would a Python dictionary, and this will return to us the same value of the User-agent field which we specified in our request object.

[Video description begins] *A code line appears in code cell 12. It reads: print(data ['user-agent']).* [Video description ends]

[Video description begins] *An output appears for code line 12. It reads: Internet Explorer/2.0.* [Video description ends]

So we have covered a few different things with regards to headers in this demo. We first saw how we can explicitly set the value of some of the fields in our headers and then use that in order to make a GET request. We also saw that the response object's headers can vary depending upon the request we have made and the URL to which we made the request, and also covered how we can access individual fields in the response headers. We also saw how we can access our original request object through the response object, and now we have also seen how we can convert the response object to the JSON format using the json function.

11. Video: Content Encoding and Binary Response Data (it_pywhpydj_01_enus_11)


Home
EncodingAndBinaryResponse
localhost:8889/notebooks/EncodingAndBinaryResponse.ipynb
Incognito
jupyter EncodingAndBinaryResponse Last Checkpoint: a minute ago (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

```
In [1]: import requests
        from pprint import pprint

In [2]: resp_obj = requests.get('http://httpbin.org/')

In [3]: resp_obj.status_code
Out[3]: 200

In [4]: resp_obj.encoding
Out[4]: 'utf-8'

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:
```

Objectives

- *define the encoding format for the contents in an HTTP Response and download and handle binary content such as images*

[Video description begins] *Topic title: Content Encoding and Binary Response Data. Your host for this session is Kishan Iyer.* [Video description ends]

So far in this course, the data which we have received within a response object, after having made a request, has been in the form of a JSON data structure, or occasionally, it has even been empty when we have made HEAD or OPTIONS request, for example.

[Video description begins] *A Jupyter notebook titled EncodingAndBinaryResponse opens. It has a menu bar with the following options: File, Edit, View, etc. There is a tool bar with the following options: Save, Add, Cut, etc. Beneath it, a set of code cells is present.* [Video description ends]

In this demo, we will see how we can handle different types of data, where we will first change the encoding of the response data, and then following that, we will also take a look at how we can handle binary data such as images. So we once again begin by importing the request and pprint modules, and then after that we will make a GET request to just httpbin.org.

[Video description begins] *Two lines of code appear in code cell 1. Code line 1 is: import requests. Code line 2 is: from pprint import pprint.* [Video description ends]

[Video description begins] *A code appears in code cell 2. It reads: resp_obj = requests.get ('http://httpbin.org/').* [Video description ends]

We will quickly confirm the status code of our response, which should be 200, and then following that, we can access the encoding field in our response object, and you should be able to see a value of utf-8.

[Video description begins] *A code appears in code cell 3. It reads: resp_obj.status_code. An output appears below it. It reads: 200.* [Video description ends]

[Video description begins] *A code appears in code cell 4. It reads: resp_obj.encoding. An output appears for code cell 4. It reads: 'utf-8 '.* [Video description ends]

This means that a response body has been encoded in the utf-8 format. But just to show you that this is a field which can be edited by us, we will now go ahead and reset this value, and on this occasion, we will set it to ISO 8859-1, which is a legacy format and is rarely used these days.

[Video description begins] *A code appears in code cell 5. It reads: resp_obj.encoding = 'ISO 8859-1 '.* [Video description ends]

However, this is just to illustrate that the encoding is something which is configurable. So now if we try to access the encoding field once more, we'll see that the value has been updated.

[Video description begins] *A code appears in code cell 6. It reads: resp_obj.encoding. An output appears for code cell 6. It reads: 'ISO 8859-1 '.* [Video description ends]

All right, so what effect does this encoding actually have? We can see that this encoding can be altered, but does this really have an effect on the data? Well, to illustrate that, we will now make another HTTP request, but this time it is going to be to github.com/timeline.json.

[Video description begins] *A code appears in code cell 7. It reads: resp_obj = requests.get (' https://github.com/timeline.json').* [Video description ends]

Now once the request has been made, we can observe the status code, and this returns a value of 410. So this is not quite a successful operation.

[Video description begins] *A code appears in code cell 8. It reads: resp_obj.status_code. An output appears for code cell 8. It reads: 410.* [Video description ends]

It doesn't matter to us, because what is of importance is the contents of the text field.

[Video description begins] *A code appears in code cell 9. It reads: pprint (resp_obj.text).* [Video description ends]

So when we view the contents of our response body, we are notified to use a different API instead.

[Video description begins] *An output appears for code cell 9. It contains a message.* [Video description ends]

Now, this doesn't really matter to us. What we are concerned with is the encoding for this particular text message which we see. Specifically, make note of the apostrophe characters. There is one on the first line in the sentence which begins with, if you're reading this, so you're is expressed as you apostrophe r-e.

[Video description begins] *In the output for code cell 9, he highlights the following text: If you're reading this, then you.* [Video description ends]

Also in the second line, you can see the word didn't, and this also includes an apostrophe character between the n and the t.

[Video description begins] *In the output for code cell 9, he highlights the following text: didn't.* [Video description ends]

So we observe the encoding which has been used in this response object, and this is utf-8.

[Video description begins] *A code appears in code cell 10. It reads: resp_obj.encoding. An output appears for code cell 10. It reads: 'utf-8 '.* [Video description ends]

Now, what if we were to change the encoding to this legacy ISO-8859-1 format?

[Video description begins] *A code appears in code cell 11. It reads: resp_obj.encoding = 'ISO 8859-1 '.* [Video description ends]

So this encoding has been updated, and now we will take a look at the response text once more, and on this occasion, you will see that the apostrophes have been replaced with a different sequence of characters.

[Video description begins] *A code appears in code cell 12. It reads: resp_obj.encoding. An output appears for code cell 12. It reads: 'ISO 8859-1 '.* [Video description ends]

[Video description begins] *A code appears in code cell 13. It reads: pprint (resp_obj.text).* [Video description ends]

[Video description begins] *An output appears for code cell 13. It contains a message. The presenter compares it with the output for code cell 9.* [Video description ends]

Now, this is just to illustrate that the encoding of a response can be edited using the request package itself. You don't need to perform some kind of post processing using some other package or in another application. So while this specific example of converting the UTF-8 encoding to ISO 8859-1 seems a little contrived. This can very much become a real problem if the data which is returned by a server happens to be in some kind of legacy format and you need to convert it to a more contemporary format, such as UTF-8. With that, we will now move along in this notebook to the handling of binary data, specifically images in a response object.

[Video description begins] *Two lines of code appear in code cell 14. Code line 1 is: from PIL import Image. Code line 2 is: from io import BytesIO.* [Video description ends]

This demo will make use of the Pillow library for Python, and this is one which is used to handle images.

[Video description begins] *A banner appears on the screen. It has the following text: Install the Pillow imaging library for Python. pip install Pillow.* [Video description ends]

If you don't have Pillow already, you can install it using pip specific leader command, pip install Pillow. With that done, you'll need to import the Image class from the Pillow module and also the BytesIO class from io. With those done, we will go ahead and make a GET request for a particular image.

[Video description begins] *A code appears in code cell 15. It reads: resp = requests.get(' https://upload.wikimedia.org/wikipedia/commons/thumb/4/4c/Moon_and_Aurora.jpg/320px-Moon_and_Aurora.jpg').* [Video description ends]

In this case, I'm requesting a specific image which is available on Wikipedia, and this is for a rather striking image of a full moon along with the aurora borealis. Now, the full link to this image is available in the code which is attached to this particular course, and once you have entered this URL in this GET request, we can run the cell, and the image should be received and available within the response object. As we typically do, we can confirm that this particular GET request was successful by examining the status code, and then following that, we examine the type of the response.

[Video description begins] *A code appears in code cell 16. It reads: resp.status_code. An output appears for code cell 16. It reads: 200.* [Video description ends]

[Video description begins] *A code appears in code cell 17. It reads: type(resp.content).* [Video description ends]

So we take a look at the content and see what the type is, and this returns a value of bytes, confirming that the response is, in fact, binary in nature.

[Video description begins] *An output appears for code cell 17. It reads: bytes.* [Video description ends]

So we have now made use of the response object in order to figure out the exact type of the contents. So now that we know it is an image, how exactly do we handle it? In our case, one option is to render the image file within our Jupyter notebook, and to do that, we make use of both the Image as well as the BytesIO class which we have imported.

[Video description begins] *A code appears in code cell 18. It reads: image = Image.open(BytesIO (resp.content)).* [Video description ends]

So here we call the function Image.open, and then to BytesIO, we pass along the binary content within a response object. This of course, is available as resp.content. So the value which is returned will be captured in this variable called image, and when we examine its type, we are shown that this is a JPEG image.

[Video description begins] *A code appears in code cell 19. It reads: type(image).* [Video description ends]

[Video description begins] *An output appears for code cell 19. It reads: PIL.JpegImagePlugin.JpegImageFile.* [Video description ends]

So now that we have this JPEG image, we can make use of this image object in order to save it as a PNG file.

[Video description begins] *A code appears in code cell 20. It reads: image.save (' aurora.png').* [Video description ends]

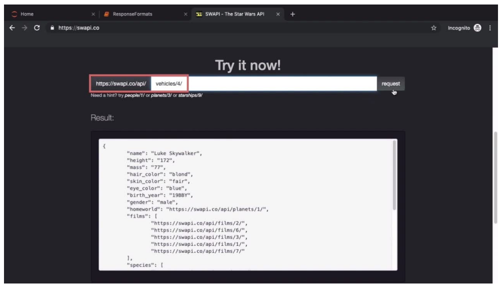
So we call the function image.save and then the argument to this is going to be, the path to the saved file. In our case, we just specify the file name, so that the file will be saved in the current working directory for the notebook. I'm going to call this aurora.png. By running the command in the cell, we will have successfully performed a GET request in order to download a JPEG file, and then we will have examined the contents using our response object and then use the Pillow library in order to confirm the file type, and also to save it down as a PNG file. So now that our file has been saved in our file system, we may as well take a look at it, and for that, we can once again, make use of the Image class.

[Video description begins] *A code appears in code cell 21. It reads: Image.open(' aurora.png').* [Video description ends]

Specifically, we make a call to Image.open, whose argument is the location of the file which we wish to render. We just pass in a relative path here to aurora.png and the image is rendered within a Jupyter notebook, and this is a rather picturesque way in which to end this demo.

[Video description begins] *An output appears for code cell 21. It contains an image.* [Video description ends]

12. Video: Handling Responses in Different Formats (it_pywhpydj_01_enus_12)



```
{
        "name": "Luke Skywalker",
        "height": "172",
        "mass": "77",
        "hair_color": "blond",
        "skin_color": "fair",
        "eye_color": "blue",
        "birth_year": "19BBY",
        "gender": "male",
        "homeworld": "https://swapi.co/api/planets/1/",
        "films": [
                "https://swapi.co/api/films/2/",
                "https://swapi.co/api/films/6/",
                "https://swapi.co/api/films/3/",
                "https://swapi.co/api/films/1/",
                "https://swapi.co/api/films/7/"
        ],
        "species": [
```

Objectives

- *identify the format of an HTTP Response and parse the data accordingly*

[Video description begins] *Topic title: Handling Responses in Different Formats. Your host for this session is Kishan Iyer.* [Video description ends]

Previously in this course, we have used the JSON function in a response object in order to view the response as a JSON data structure and then parse it if necessary.

[Video description begins] *A jupyter notebook titled ResponseFormats opens. It has a menu bar with the following options: File, Edit, View, etc. There is a tool bar with the following options: Save, Add, Cut, etc. Beneath it, a set of code cells is present.* [Video description ends]

However, one important thing to note is that this JSON function can only be used if the response object does actually contain JSON data. This is what we will illustrate in this example, where we will also show that the JSON function can work even if the HTTP request was unsuccessful, as long as the return data is in JSON format, even if it includes an error message.

So we begin coding up this new jupyter notebook called ResponseFormats, and we will first import the libraries which we need, and then following that, we will make a GET request to the Star Wars API site, and before invoking the GET request from our notebook, we can open up a new tab in our browsers, and then navigate to the Star Wars API, which is available at swapi.co.

[Video description begins] *Three lines of code appear in code cell 1. Code line 1 is: import requests. Code line 2 is: from pprint import pprint. Code line 3 is: import json.* [Video description ends]

[Video description begins] *He opens a new tab in the browser.* [Video description ends]

[Video description begins] *In the address bar, he types https://swapi.co. He then presses Enter.* [Video description ends]

From here, we scroll a little further down, and then we can enter the URL of the API call which we need to make.

[Video description begins] *A new web page titled SWAPI - The Star WarsAPI opens. It has three options at the top: Home, About, and Documentation. An input box is present next to the following URL: https://swapi.co/api/. The input box contains the following text: people/1/. A request button is present beside the input box. Beneath it, there is a Result text box. He clicks on the input box. The text inside it changes to: vehicles/4/.* [Video description ends]

In this case, we are searching for the details of vehicle number 4 in this database. Upon hitting the request button, the details for the vehicle get populated in the field below, and we can see that this is a vehicle called a Sand Crawler and more importantly for us, we can also confirm that the result is in a JSON format.

[Video description begins] *He clicks the request button.* [Video description ends]

[Video description begins] *He highlights the entire text in the Result text box. He highlights the following text in the Result text box: "name": "Sand Crawler",.* [Video description ends]

So we expect that we can call the JSON function on the response object when we invoke a request from our notebook. So we head back to the notebook, and then make this GET request call to the Star Wars API and search for vehicle number 4.

[Video description begins] *He closes the SWAPI tab. The jupyter notebook tab is open.* [Video description ends]

[Video description begins] *A code line appears in code cell 2. It reads: resp_obj = requests.get ('https://swapi.co/api/vehicles/4/ ').* [Video description ends]

We then check the status code in our response object, and as you would expect this returns a value of 200.

[Video description begins] *A code line appears in code cell 3. It reads: print(resp_obj.status_code).* [Video description ends]

[Video description begins] *An output appears for code cell 3. It reads: 200.* [Video description ends]

So we now know that we have JSON data and that the request was successful. At this point, we can invoke the JSON function on the response object, knowing that we do have JSON data to work with.

[Video description begins] *A code line appears in code cell 4. It reads: pprint (resp_obj.json()).* [Video description ends]

So using the pretty print function to print out the JSON data, we get the same details for the Sand Crawler vehicle.

[Video description begins] *An output appears for code cell 4. It contains the following information: 'consumables': '2 months', 'name': 'Sand Crawler', 'model': 'Digger Crawler', etc.* [Video description ends]

The pretty print function has sorted all of the fields in alphabetical order, which is why the ordering of the fields here is a little different from what we saw on the website. Now, in this particular case we were able to view the JSON data which was returned by using the API on that website.

However, how do we know for sure what the contents of the response is? Specifically, how do we know that it contains JSON data before we invoke the JSON function? Not all APIs will have a nice website available as the Star Wars API does. Well, we can just examine the headers in the response object, specifically the field content-type.

[Video description begins] *A code line appears in code cell 5. It reads: resp_obj.headers ['content-type'].* [Video description ends]

So in this case, when we examine the content-type, this returns the value of application/json. So we know that the response does contain JSON data, and that it will be safe for us to execute the JSON function.

[Video description begins] *An output appears for code cell 5. It reads: ' application/json'*. [Video description ends]

But what happens if the response does not contain JSON information? To demonstrate that, we make one more GET request, and on this occasion we make it to yahoo.com.

[Video description begins] *A code line appears in code cell 6. It reads: resp = requests.get ('https://www.yahoo.com')*. [Video description ends]

So this response object has been captured and when we take a look at the status code, the value of 200 confirms that our operation was a success.

[Video description begins] *A code line appears in code cell 7. It reads: print(resp.status_code)*. [Video description ends]

[Video description begins] *An output appears for code cell 7. It reads: 200*. [Video description ends]

So does this mean that we can execute the JSON function on the response?

[Video description begins] *A code line appears in code cell 8. It reads: pprint (resp.json())*. [Video description ends]

So we just run this here and we get an error.

[Video description begins] *A JSONDecodeError appears in the output for code cell 8*. [Video description ends]

In fact, we get a JSONDecodeError as you can see from the message at the bottom, and the reason for this is that the return value is not in JSON format.

[Video description begins] *He highlights the following message at the bottom of the error: Expecting value: line 1 column 1 (char 0)*. [Video description ends]

If you're not sure what kind of data to expect in return, we need to perform a check of the content type in the response header. In this specific case, the value of text/html will have informed us that the data is in HTML format, and since a GET request was to yahoo.com, which is the Yahoo Homepage, this is exactly what we should expect.

[Video description begins] *A code line appears in code cell 9. It reads: resp.headers ['content-type']*. [Video description ends]

[Video description begins] *An output appears for code cell 9. It reads:'text/html; charset= UTF-8 '*. [Video description ends]

So we have now covered two different examples of the JSON function call on the response object. When there is JSON data to work with, the function executes smoothly and returns a JSON object.

[Video description begins] *The following command appears on the screen: pprint (resp_obj.json())*. [Video description ends]

However, if the response does not contain JSON information, it throws this JSONDecodeError. Now, we will go ahead and make one more GET request and this time, it is to googleapis.com.

[Video description begins] *The following command appears on the screen:pprint (resp.json()). Both the commands disappear from the screen. A code line appears in code cell 10. It reads: resp_obj = requests.get ('http://maps.googleapis.com/maps/api/geocode/json')*. [Video description ends]

Specifically, this is a request to the Google Maps API, and at least as of this recording, a GET request to this URL does return JSON data in the response. So I'll just go ahead and run the cell, and then from the response object will examine the status code.

[Video description begins] *A code line appears in code cell 11. It reads: resp_obj.status_code*. [Video description ends]

So the value of 400 implies that the request which we sent to the Google Maps API was not correct.

[Video description begins] *An output appears for code cell 11. It reads: 400*. [Video description ends]

Now does this mean that calling the JSON function on the response object will result in an error?

[Video description begins] *A code line appears in code cell 12. It reads: resp_obj.json()*. [Video description ends]

Well, when we run this cell, there is no JSONDecodeError.

[Video description begins] *An error message appears in the output for code cell 12. It contains the following information: Missing the 'address', 'components', etc*. [Video description ends]

In fact, what we see is an error message along with an invalid request status code. So this is confirmation of the fact that the JSON function will be able to parse any returned data, as long as it is in the JSON format. It doesn't matter whether this includes an error message or if the response status code points to an error.

In any case, in order to ensure that the call to the JSON function will not result in an error. You can always examine the content type field in the header before executing that function. So now that we are comfortable with handling JSON data in a response, we will now examine how we can handle data in its raw format.

[Video description begins] *A code line appears in code cell 13. It reads: resp_obj = requests.get ('https://swapi.co/api/vehicles/4/',stream=True).* [Video description ends]

For this, we make one more GET request to the Star Wars API website, and just as we did previously, this is for vehicle number 4 in their database. The difference in this GET request however, is that we are setting the value of the stream argument to true. This argument is typically used if we have a very large file which needs to be downloaded, and rather than have the file loaded into memory, it can be streamed directly onto your file system.

To see how this works, we make this GET request, and then following that, if you'd like to view the raw socket response which came in from the server, we can access the property raw of the response object. So this confirms that this contains an HTTP response.

[Video description begins] *A code line appears in code cell 14. It reads: resp_obj.raw.* [Video description ends]

But to view the actual raw contents, we will call the read function on this raw object.

[Video description begins] *An output appears for code cell 14. It reads: <urllib3.response.HTTPResponse at 0x109470748>.* [Video description ends]

[Video description begins] *A code line appears in code cell 15. It reads: resp_obj.raw.read(10).* [Video description ends]

The argument of 10 to the read function implies that we wish to read the first 10 characters in that raw object, and this returns to us in byte form the first 10 characters of our response.

[Video description begins] *An output appears for code cell 15. It reads: b'\ xlf \ x8b \ x08 \ x00 \ x00 \ x00 \ x00 \ x00 \ x00 \ x03'.* [Video description ends]

I had mentioned that setting the stream argument to true when making a get request, is often used to download large files and stream it directly to the file system, and one way to do it is illustrated right here, where we make use of a width block in order to make a GET request and capture the response.

[Video description begins] *The command in code cell 13 appears on the screen.* [Video description ends]

[Video description begins] *The command disappears.* [Video description ends]

[Video description begins] *Four code lines appear in code cell 16. Code line 1 is: with requests.get ('https://swapi.co/api/vehicles/4/',stream=True) as response:. Code line 2 is: with open ('raw_file.txt', 'wb') as b:. Code line 3 is: for chunk in response.iter_content (1000):. Code line 4 is: b.write(chunk).* [Video description ends]

We then open up a file on our file system to which we will write the details in the response, and then we can iterate through all of the contents in the response in chunks by using the response object's iter_content function. The argument of 1000 here means that the size of each chunk will be 1,000 bytes, and we then use the write function of a file in order to write those 1,000 bytes in each iteration to the file.

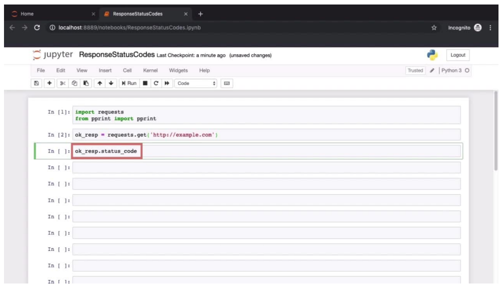
So with such a block, we can ensure that only 1,000 bytes of the response is loaded into memory at any given point in time, and when we run the cell, once the file has been written, we can examine its contents by running the cat command from our Jupyter Notebook, and the file does contain the same JSON data which we saw previously.

[Video description begins] *A code line appears in code cell 17. It reads: !cat 'raw_file.txt'.* [Video description ends]

[Video description begins] *An output appears for code cell 17. It contains the following information: "name" : "Sand Crawler", "model" : "Digger Crawler", etc.* [Video description ends]

So we have conserved memory by streaming the response object directly to a file, in 1,000-byte chunks.

13. Video: HTTP Status Codes (it_pywhpydj_01_enus_13)



```
In [1]: import requests
        from pprint import pprint

In [2]: ok_resp = requests.get('http://example.com')

In [ ]: ok_resp.status_code
```

Home
ResponseStatusCodes
localhost:8889/notebooks/ResponseStatusCodes.ipynb
Incognito
jupyter ResponseStatusCodes Last Checkpoint: a minute ago (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

Objectives

- *handle successful and unsuccessful HTTP requests according to the status code of the response*

[Video description begins] *Topic title: HTTP Status Codes. Your host for this session is Kishan Iyer.* [Video description ends]

So far in this course, we have made several types of requests, and on most occasions, we have examined the status code in the returned response object.

[Video description begins] *A jupyter notebook titled ResponseStatusCodes opens. It has a menu bar with the following options: File, Edit, View, etc. There is a tool bar with the following options: Save, Add, Cut, etc. Beneath it, a set of code cells is present.* [Video description ends]

Now on some occasions we have seen that the response code is not 200, which denotes a successful message, and in this demo, we will see different ways in which we can handle an error response. So we begin by importing the requests as well as the pprint modules into our code, and then we will make a GET request which will return a 200 status code.

[Video description begins] *Two lines of code appear in code cell 1. Code line 1 is: import requests. Code line 2 is: from pprint import pprint.* [Video description ends]

[Video description begins] *A code line appears in code cell 2. It reads: ok_resp = requests.get ('http://example.com').* [Video description ends]

So this is to example.com. So once we make this GET request, we confirm that the status code is 200, and now that we have that confirmation, we can also examine another attribute in the response object, and this is called ok.

[Video description begins] *A code line appears in code cell 3. It reads: ok_resp.status_code.* [Video description ends]

[Video description begins] *An output appears for code cell 3. It reads: 200.* [Video description ends]

[Video description begins] *A code line appears in code cell 4. It reads: ok_resp.ok.* [Video description ends]

This will return a value of true if everything was fine with the request, and this is a field which is used in order to get a binary value for the status code of true or false rather than an entire range of status codes which are usually much more specific.

[Video description begins] *An output appears for code cell 4. It reads: True.* [Video description ends]

So now that we have made a valid GET request, we will make an invalid one, one which we know will generate an error.

[Video description begins] *A code line appears in code cell 5. It reads: bad_resp = requests.get ('https://yahoo.com/alf2adfd5').* [Video description ends]

Here, I'm just attaching some garbage value after yahoo.com, so that we are issuing a GET request to a URL which does not exist. So we run this code and no error is thrown.

[Video description begins] *A code line appears in code cell 6. It reads: bad_resp.status_code.* [Video description ends]

But on examining the status code, this gives a value of 404.

[Video description begins] *An output appears for code cell 6. It reads: 404.* [Video description ends]

If you have used the Internet often enough, you will know that this is an error code which represents that the page does not exist. However, there are multiple forms of errors which can be thrown when you make a GET request, and when you encounter such an error in your code, there are multiple ways in which you might handle it. For one, you may want all of the details of the error to be presented, and if you wish to do that, there is a function which you can use in the response object, and this is called raise_for_status.

[Video description begins] *A code line appears in code cell 7. It reads: bad_resp.raise_for_status().* [Video description ends]

So upon running this, an error will be thrown and all of the details corresponding to that error will also be available.

[Video description begins] *An error appears in the output for code cell 7.* [Video description ends]

In this specific example, we can see that this was an HTTP error and that this corresponds to a message of URL not found.

[Video description begins] *He highlights the following part of the error: HTTPError: 404 Client Error: Not Found for url: https://www.yahoo.com/alf2adfd5.* [Video description ends]

If you are making GET request in your own application, it may be best for you to raise an exception and then handle it in your own way. Since there are multiple status codes which correspond to an error and also several different status codes which correspond to a successful GET request, you could either use the ok attribute in the response object to find out whether your GET request was valid or invalid, or in the case of an error, if you would like more details available, then you can raise an exception using the raise_for_status function. Now what happens if you were to call this raise_for_status function using a response object returned from a valid GET request?

[Video description begins] *A code line appears in code cell 8. It reads: ok_resp.raise_for_status(). The following command appears on the screen: ok_resp = requests.get ('http://example.com').* [Video description ends]

So from a valid call to example.com, we have captured the response in this variable ok_resp. So we call the raise_for_status using that object and on this occasion, there is no data which is returned, and importantly, no error is thrown.

[Video description begins] *The command disappears from the screen*. [Video description ends]

Now, we have already taken a look at the headers of a response object previously in this course, but now we can examine the type of the headers.

[Video description begins] *A code line appears in code cell 9. It reads: type(ok_resp.headers)*. [Video description ends]

So we use the Python type function to get the type here, and what we find is that the headers are in fact a special data structure called a case insensitive dictionary.

[Video description begins] *An output appears for code cell 9. It reads: requests.structures.CaseInsensitiveDict*. [Video description ends]

So if we print out the headers, we can see that this does correspond to a Python dictionary, in that there are keys and values.

[Video description begins] *A code appears in code cell 10. It reads: pprint (ok_resp.headers)*. [Video description ends]

[Video description begins] *An output appears for code cell 10. It contains the following information: Content-Encoding, Accept-Ranges, Content-Type, etc*. [Video description ends]

But the difference between a standard Python dictionary and this case insensitive dictionary is that you can access the values without worrying about the case of the keys. So for example, if we wish to access the value corresponding to the Content-Type key, we can use this format, which is similar to a Python dictionary, and this returns the value of text/html, along with the character set which is in use.

[Video description begins] *A code line appears in code cell 11. It reads: ok_resp.headers ['Content-Type']*. [Video description ends]

[Video description begins] *An output appears for code cell 11. It reads:'text/html; charset= UTF-8* '. [Video description ends]

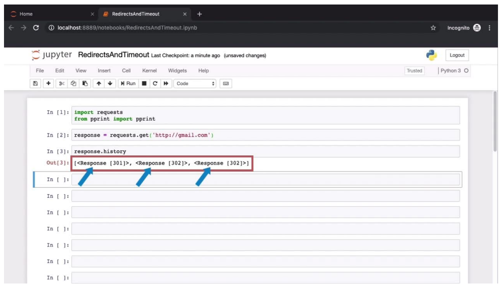
Now, if you were to change the C and the T to lowercase, this will still return the exact same value. Now, another thing to keep in mind is that there is another function which you can use with a case insensitive dictionary in order to return the value corresponding to a key. This is the GET function.

[Video description begins] *A code line appears in code cell 12. It reads: ok_resp.headers.get ('Content-Type')*. [Video description ends]

So rather than accessing the values using the square brackets, you can instead call the GET function and then specify the key whose corresponding value you wish to retrieve. So we run this to get the value for Content-Type, and this, again, returns a value of text/html.

[Video description begins] *An output appears for code cell 12. It reads:'text/html; charset= UTF-8* '. [Video description ends]

14. Video: Redirects and Timeouts (it_pywhpydj_01_enus_14)


Home
RedirectsAndTimeout
localhost:8889/notebooks/RedirectsAndTimeout.ipynb
Incognito
jupyter RedirectsAndTimeout Last Checkpoint: a minute ago (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

```
In [1]: import requests
        from pprint import pprint

In [2]: response = requests.get('http://gmail.com')

In [3]: response.history
Out[3]: [<Response [301]>, <Response [302]>, <Response [302]>]

In [ ]:
```

Objectives

- *recognize when an HTTP request has triggered a redirect, explore the redirect history, and configure the read and connect timeout values for a request*

[Video description begins] *Topic Title: Redirects and Timeouts. Your host for this session is Kishan Iyer.* [Video description ends]

If you are building a website, either for the company you are working for or even for yourself, you may implement redirects in order to direct different users to different versions of the website. For example, a global media company may have a .com domain which everyone in the world will access, but what will happen behind the scenes is that they will be redirected to their own local version of the website.

[Video description begins] *A jupyter notebook page with the heading 'Redirects and Timeout' is open. It has a Logout button on the top-right corner. A menu bar is present with the following options: File, Edit, View, Insert, etc. There is a toolbar with the following buttons: Save, Insert cell, Run, etc. Beneath the buttons, several code cells are present*. [Video description ends]

So, if you are building such a web application, you may wish to check in a programmatic manner whether the redirects have been set up correctly. To do that, you can make use of the requests package in Python. We can see how exactly this works where we can get a full history of all of the redirects which have been implemented. So we begin as usual by importing requests and pprint, and the first request which we make is a GET request to gmail.com.

[Video description begins] *The following code appears in code cell 1: import requests from pprint import pprint.* [Video description ends]

Now, this will return a response, and if you wish to see whether any redirects were involved, we can take a look at the history attribute in the response object, and in my case, I can see that there were in fact three different steps involved for my GET request to gmail.com.

[Video description begins] *The following code appears in code cell 2: response = requests.get('http://gmail.com').* [Video description ends]

[Video description begins] *The following code appears in code cell 3: response.history.* [Video description ends]

There is a response object corresponding to each of these steps which have been captured in this history attribute.

[Video description begins] *The following output appears for code cell 3: [<Response[301]>, <Response [302]>, <Response[302]>].* [Video description ends]

Do keep in mind though, that depending upon your geographical location, the number of redirects, if any, will vary for gmail.com, so you may not see the three different responses in your history as I do. One more observation which you can make about the responses in the history here is that you can see the status codes for each of the responses. These are 301, 302, and 302, in my case, and each of these correspond to redirects. However, when taking a look at the overall status_code for my response, this returns a value of 200. So the request which we made was a GET request to gmail.com, however, what is the URL corresponding to the response?

[Video description begins] *The following code appears in code cell 4: response.status_code. 200 appears as the output.* [Video description ends]

[Video description begins] *He highlights the following code in code cell 2: requests.get('http://gmail.com').* [Video description ends]

What is the final URL which I have been redirected to? This can be obtained from the URL property in the response object, and you can see this rather long URL corresponding to accounts.google.com. Now I'm currently logged in to Gmail, which is why such a URL is visible.

[Video description begins] *The following code appears in code cell 5: response.url. The following URL appears as the output: 'https://accounts.google.com/ServiceLogin?service=mail&passive=true&rm=false&continue=https://mail.google.com/mail/&ss=1&scc=1&ltmpl=default&ltmplcache=2&emr=1&osid=1'.* [Video description ends]

It is possible that you may see something a little different. So we initially made a GET request to gmail.com, and now we know the final destination of our request. From our response history, we know that there were multiple steps involved in reaching this final destination. But if you wish to list down all of the different URLs which were hit in this process, we can write some code in order to iterate through each of the responses in the response history. So first, we check that the response.history attribute is not null, and if it does contain values, then we will print out the Redirect history.

[Video description begins] *Ten lines of code appear in code cell 6. Code line 1 reads: if response.history: . Code line 2 is: print ("Redirect history: "). Code line 3 is: for resp in response.history:. Code line 4 is: print (resp.status_code, resp.url). Code line 6 is: print ("\nFinal destination:"). Code line 7 is: print (response.status_code, response.url). Code line 9 is: else:. Code line 10 is: print ("Request was not redirected").* [Video description ends]

[Video description begins] *He highlights code line 1 and talks about it.* [Video description ends]

[Video description begins] *He highlights code line 2 and talks about it.* [Video description ends]

To do that, we iterate through each of the individual responses in response.history, and then we print out the status code corresponding to each of those responses, along with the response URL.

[Video description begins] *He highlights code line 3 and talks about it.* [Video description ends]

[Video description begins] *He highlights code line 4 and talks about it.* [Video description ends]

At the end of this for loop we will just print out the final destination of our GET request, where we get the overall status code from the response object, and also print out response.url.

[Video description begins] *He highlights code line 6 and talks about it.* [Video description ends]

[Video description begins] *He highlights code line 7 and talks about it.* [Video description ends]

However, if response.history was empty, that is there was no redirection involved, then we print out the fact that the request was not redirected.

[Video description begins] *He highlights code lines 9 and 10 and talks about them.* [Video description ends]

So when we run the cell, we can see the entire redirect history.

[Video description begins] *Three lines of output appear for the heading Redirect history. Line 1 reads: 301 http://gmail.com/. Line 2 is: 302 https://www.google.com/gmail/. Line 3 is: 302 https://mail.google.com/mail/. The following URL appears as output under the heading Final destination: 200 https://accounts.google.com/ServiceLogin?service=mail&passive=true&rm=false&continue=https://mail.google.com/mail/&ss=1&scc=1&ltmpl=default&ltmplcache=2&emr=1&osid=1.* [Video description ends]

Again, in my case, given I'm located in India, there were three different jumps involved. The first request was to http://gmail.com, and this was redirected to google.com/gmail, and then to mail.google.com.

[Video description begins] *He points to output line 1 and talks about it.* [Video description ends]

[Video description begins] *He points to output line 2 and talks about it. He points to output line 3 and talks about it.* [Video description ends]

Before the final destination of accounts.google.com when that long URL was hit.

[Video description begins] *He highlights the url under the heading Final destination and talks about it.* [Video description ends]

So, as a web developer, if you wish to ensure that your redirect logic has been implemented according to your specifications, then you can examine the entire redirect history in this manner. We move along now and examine a property called is_redirect corresponding to the response object. So we do know that our request was redirected a few times.

[Video description begins] *The following code appears in code line 7: response.is_redirect.* [Video description ends]

However, this variable response corresponds to the final response object and not one of the intermediate responses, which were redirected, which is why the value of is_redirect for this response variable is false.

[Video description begins] *False appears as the output.* [Video description ends]

There is also a property called is_permanent_redirect, which shows whether

[Video description begins] *The following code appears in code line 8: response.is_permanent_redirect.* [Video description ends]

this response represents a permanent redirect, and this also is false. So clearly this final response object does not represent a redirect.

[Video description begins] *False appears as the output.* [Video description ends]

However, if we iterate through each of the individual responses in our history, we can access the is_redirect property, for example, for the first of those responses, and this returns a value of True.

[Video description begins] *The following code appears in code line 9: response.history[0].is_redirect.* [Video description ends]

[Video description begins] *True appears as the output.* [Video description ends]

You can try the same thing for all of the other responses in your history. Now, what if you wish to disable redirects, and just hit the first URL to which you're making the GET request? Well for that, you can once again make a GET request, and then specify the value of the allow_redirects argument to be false.

[Video description begins] *The following code appears in code line 10: response = requests.get('http://google.com' , allow_redirects = False). He highlights the following part of the code: allow_redirects = False).* [Video description ends]

Now this will disable redirects, and note that on this occasion our get request is to google.com and not Gmail.

[Video description begins] *He highlights the following part of the code and talks about it: ('http://google.com' ,.* [Video description ends]

So when you run this, you can examine the status code in the response, and this returns a value of 301, and this in fact, corresponds to the first response object which you would see in the response history if you enable redirects.

[Video description begins] *The following code appears in code line 11: response.status_code.* [Video description ends]

[Video description begins] *301 appears as the output.* [Video description ends]

If you examine the response.history here, what it returns to us is a blank list.

[Video description begins] *The following code appears in code line 12: response.history. The following output appears: [ ].* [Video description ends]

Now, we can also issue a head request to google.com, and on this occasion, we specify the value of allow_redirects to be equal to True.

[Video description begins] *The following code appears in code line 13: resp_head = requests.head('http://google.com' , allow_redirects = True).* [Video description ends]

In the case of a GET request, the default value for allow_redirects is True, but is actually false in the case of a head request. So when we run this and then examine the status code, we are returned a value of 200.

[Video description begins] *The following code appears in code line 14: resp_head.status_code.* [Video description ends]

We will now issue the same head request, but we will not specify the value of allow redirects, which should by default, be set to false.

[Video description begins] *200 appears as the output.* [Video description ends]

[Video description begins] *The following code appears in code line 15: .resp_head = requests.head('http://google.com').* [Video description ends]

So when we examine the status code on this occasion, we are returned a value of 301, corresponding to a redirect.

[Video description begins] *The following code appears in code line 16: resp_head.status_code.* [Video description ends]

So if your aim is just to confirm whether a redirect has been implemented, then you can make a head request, and then check the status code, or the is_redirect property.

[Video description begins] *301 appears as the output.* [Video description ends]

But to get the details for all of the individual hops involved in the redirects you will need to make a GET request, and then examine the response history. We now move along to one more feature in the Python request package, and this is where we can check whether a particular page is taking a really long time to respond.

[Video description begins] *The following code appears in code line 17: requests.get ('http://github.com' timeout=0.001).* [Video description ends]

So here, we make a GET request to github.com, and then specify a timeout value of 0.001 second, or one millisecond.

[Video description begins] *He highlights the following part of the code and talks about it:'http://github.com'.* [Video description ends]

This will raise an exception if no bytes are received at the socket within one millisecond.

[Video description begins] *He highlights the following part of the code and talks about it: timeout=0.001).* [Video description ends]

Note that this timeout value does not correspond to the time taken to download the entire response, it just represents the time taken to receive any kind of response from the server. Now this is a deliberately low value for the timeout to guarantee a timeout error will be thrown. So I'm just going to run this, and then we can observe exactly what a timeout exception looks like. So the single timeout value which we specified corresponded to a connection timeout value, that is the amount of time taken to receive any response from the server, and not a read timeout value which represents the time taken to download the full response.

[Video description begins] *Several lines of output appear. He scrolls down the output and selects the following lines: ConnectTimeout: HTTPConnectionPool(host='github.com' , port=80): Max retries exceeded with url: / (Caused by Connect TimeoutError(<urllib3.connection.HTTPConnection object at 0x113061e80>, 'Connection to github.com timed out. (connect timeout=0.001) ')).* [Video description ends]

Now if you wish to examine the response time for a server, both in terms of connection time as well as the time taken to download the response, you can specify both the connection timeout value and the read timeout value as a tuple.

[Video description begins] *The following code appears in code line 18: requests.get('https://github.com' timeout=(5, 18)).* [Video description ends]

[Video description begins] *He highlights the following part of the code and talks about it: timeout=(5, 18)).* [Video description ends]

So here we represent values of 5 seconds and 18 seconds for the connection and read timeouts, and we make our GET request to github.com, and on this occasion we are returned a response code of 200.

[Video description begins] *He highlights the following part of the code and talks about it: requests.get('https://github.com.* [Video description ends]

Now, if you like to wait indefinitely in order to establish a connection with an endpoint, then you can set a value of none for the timeout argument.

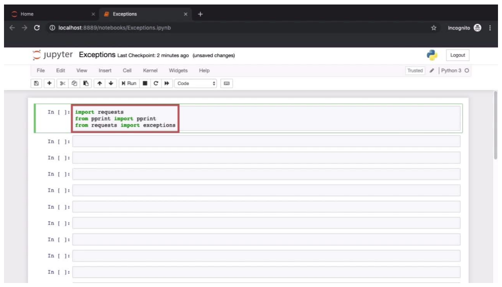
[Video description begins] *The following output appears in output line 18: <Response [200]>.* [Video description ends]

[Video description begins] *The following code appears in code line 19: requests.get ('https://github.com', timeout=None).* [Video description ends]

So when we run this for github.com, we are very quickly returned a response with a status code of 200.

[Video description begins] *The following output appears in output line 19: <Response [200]>*. [Video description ends]

15. Video: Exceptions (it_pywhpydj_01_enus_15)


jupyter
Exceptions Last Checkpoint: 2 minutes ago (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help

```
import requests
from pprint import pprint
from requests import exceptions
```

Objectives

- *catch errors that are thrown by HTTP requests by using the exceptions module in the Requests package*

[Video description begins] *Topic title: Exceptions. Your host for this session is Kishan Iyer.* [Video description ends]

So far in this course, we have covered a few instances where an error was thrown when we tried to issue a request to a URL.

[Video description begins] *A jupyter notebook page with the heading Exceptions is open. It has a Logout button on the top-right corner. A menu bar is present with the following options: File, Edit, View, Insert, etc. There is a toolbar with the following buttons: Save, Insert cell, Run, etc. Below the buttons, several code cells are present*. [Video description ends]

Now, there are many different reasons why an error might be thrown, but in this demo, we will show you a couple of examples of how such an error can be caught and handled. To do that, we will import both the requests and pprint modules once more.

[Video description begins] *Three lines of code appear in code cell 1. Code line 1 reads: import requests. Code line 2 is: from pprint import pprint. Code line 3 is: from requests import exceptions*. [Video description ends]

But on this occasion, we will also import the exceptions module within the requests package. The exceptions package includes definitions for a number of different exceptions which we can catch and then handle in a manner which we see fit for our application. So we begin by making a get request to a URL which does not exist.

[Video description begins] *The following code appears in code cell 2: requests.get('http://nonexistent.com/')*. [Video description ends]

So I'm making this request to nonexistent.com which, at least at the time of this recording, does not represent a valid URL. So when we run this, as expected, an error has been thrown, and in order to figure out exactly what kind of error has been thrown here, we can scroll all the way to the bottom, and we can see that this has thrown something called a ConnectionError.

[Video description begins] *Several lines of output appear for code cell 2. He scrolls down the output and selects the following part: ConnectionError: HTTPConnectionPool(host='nonexistent.com' , port=80): Max retries exceeded with url: / (Caused by NewConnectionError('<urllib3.connection.HTTPConnection object at 0x113c1f7f0>: Failed to establish a new connection: [Errno 8] nodename nor servname provided, or not known '))*. [Video description ends]

Now, in your code, you may wish to catch exactly such errors and may have specific logic to handle such connection errors. So if this is the exact type of error which you wish to catch and handle, then you can implement a try and except block in Python, and then within the except block, the specific error which you wish to catch is an exceptions.ConnectionError, and in this particular example, all we're doing is printing out the fact that an connection error was thrown, but of course, in your own application, you may implement your own logic to handle such errors.

[Video description begins] *Four lines of code appear in code cell 3. Code line 1 reads: try: requests.get('http://nonexistent.com'). Code line 3 is: except exceptions.ConnectionError:. Code line 4 is: print("Error: Connection error")*. [Video description ends]

[Video description begins] *In code cell 3, he highlights code lines 3 and 4 and talks about them*. [Video description ends]

So when we run this, the error which is thrown is caught, and we don't see the entire stack tray for the error but just the print statement which we implemented within the except block.

[Video description begins] *The following output appears for code cell 3: Error: Connection error.* [Video description ends]

All right, so this represents one of the errors which we have caught previously in this course, and we now move along to a different type of error which we saw in the previous video, which is a connection timeout error.

[Video description begins] *The following code appears in code cell 4: requests.get ('http://github.com' , timeout=0.01).* [Video description ends]

So we issue a GET request to github.com, and we set a connection timeout value of 0.01 seconds, which at least for me is guaranteed to throw an error. So when running this, sure enough, the error has been thrown, and scrolling further down, I can see that the specific error is a ConnectTimeout.

[Video description begins] *Several lines of output appear for code cell 4. He scrolls down the output and selects the following part: ConnectTimeout: HTTPConnectionPool(host='github.com' , port=80): Max retries exceeded with url: / (Caused by Connect TimeoutError(<urllib3.connection.HTTPConnection object at 0x113d461d0>, 'Connection to github.com timed out. (connect timeout=0.01) ')).* [Video description ends]

This error has been defined within the exceptions module, and we can implement one more try except block in order to catch this exception and then handle it in a manner we see fit for our application.

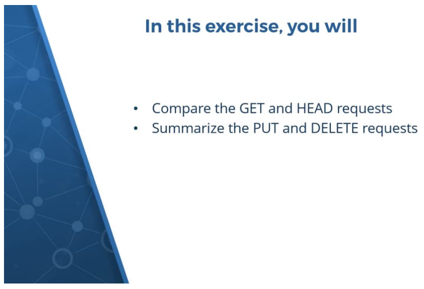
[Video description begins] *Four lines of code appear in code cell 5. Code line 1 reads: try: requests.get ('http://github.com' , timeout=0.001). Code line 3 is: except exceptions.ConnectTimeout:. Code line 4 is: print(": Timeout Error").* [Video description ends]

So when running this, this prints out Timeout Error just as we have defined in our try except block.

[Video description begins] *The following output appears for code cell 5: Timeout Error.* [Video description ends]

So now we know that we can make use of the exceptions module within the request package in order to catch specific types of exceptions and then handle them within our code.

16. Video: Exercise: HTTP Requests with Python (it_pywhpydj_01_enus_16)


In this exercise, you will

- Compare the GET and HEAD requests
- Summarize the PUT and DELETE requests

Objectives

- *compare the GET and HEAD requests, summarize the PUT and DELETE requests, and view the redirect history following a GET request*

[Video description begins] *Topic title: Exercise: HTTP Requests with Python. Your host for this session is Kishan Iyer.* [Video description ends]

In this exercise, you will start off by comparing GET and HEAD requests. During this course, there were many different types of request which we covered, and out of those, GET and HEAD requests did have a few points in common, but there were also a few notable differences which we took a look at. You will recollect exactly what those commonalities and differences are. Next, you'll summarize both PUT and DELETE requests.

Both of these types of requests are meant to carry out very specific operations, and you'll need to identify exactly what those are, and finally, you'll write a couple of lines of code, firstly to invoke a GET request and then to view the redirect history for that GET request. During one of the demos in this course, we saw how a single GET request may trigger a number of different redirects before you finally get a response object.

You will need to recall how we can use the request package in Python in order to make the request and then view the redirect history. All of these topics were covered during this course, so please pause this video, and then take some time to perform this exercise on your own. For the first task in the exercise, you needed to compare GET and HEAD requests.

[Video description begins] *A Solution banner appears.* [Video description ends]

So both GET and HEAD request do have a lot of points in common, specifically they're used to get information from a server. So other than that one common point, when you invoke such a request, the headers which are returned in the response object will be identical.

Also, both GET and HEAD requests are idempotent operations, which means that invoking them multiple times will return the same value, and in fact, both of these are effectively read operations where they don't cause any change on the server. Moving along now to the difference in the response bodies is that the HEAD request in fact does not have a body at all.

This is because the HEAD request is meant to be used to get the metadata for a resource. While the GET request is used to get not just the metadata, but also the data itself. For example, you can use a HEAD request in order to check for the existence of a resource before you initiate a download, and also the size and format of the data which will be returned, before actually triggering a GET request to get that data.

Moving along now to the next task, which was to summarize the PUT and DELETE requests. In the case of a PUT request, this is one which is used to either create or replace a value or resource at the target. In fact, a PUT request is often compared to a POST request, since both cases involve submitting data to the server. However, the significant difference between the two is that a PUT request is idempotent in nature.

That is, it'll either create a resource which does not exist or it'll entirely replace the existing resource, and invoking the same PUT request multiple times, will produce the exact same results. Moving along now to a DELETE request, and this is simply used in order to delete a file or resource at the specified URL endpoint. For the final task in the exercise, you needed to write a couple of lines of code, in order to first invoke a GET request and then view the redirect history for that request, if there was any. Well, to first create a GET request, you can use the requests module's get function.

[Video description begins] *The following command appears: response = requests.get (<url>, <options>).* [Video description ends]

The argument to this include the url to which you will submit the GET request along with any options, if there are any. So once this GET request has been made, this will return a response object, and using that response object you can get the redirect history, specifically it is available in the history attributes of the response object.

[Video description begins] *The following command appears: response.history.* [Video description ends]

As we saw in this course, this is in fact, a list of response objects, where each response object represents a single redirect.

Course File-based Resources

- HTTP Requests with Python: Installing the Requests Package
  Assets
- HTTP Requests with Python: A Basic GET Request
  Assets
- HTTP Requests with Python: Exploring an HTTP Response Containing JSON Data
  Assets
- HTTP Requests with Python: Including Parameters in a GET Request
  Assets
- HTTP Requests with Python: A Basic POST Request
  Assets
- HTTP Requests with Python: A POST Request with Multiple Parameters
  Assets
- HTTP Requests with Python: The HEAD Request
  Assets
- HTTP Requests with Python: The PUT, OPTIONS, and DELETE Requests
  Assets
- HTTP Requests with Python: Working with Request and Response Headers
  Assets
- HTTP Requests with Python: Content Encoding and Binary Response Data
  Assets
- HTTP Requests with Python: Handling Responses in Different Formats
  Assets
- HTTP Requests with Python: HTTP Status Codes
  Assets

- HTTP Requests with Python: Redirects and Timeouts
  Assets

- HTTP Requests with Python: Exceptions
  Assets