

Creating URL query strings in Python

How to make Python do the tedious work of creating URL query strings.

Summary

Most Web APIs require you to pass in configuration values via a URL query string. Creating these strings is a matter of reading the API's documentation, and then either doing the mind-numbing work of manually creating the query strings. Or using Python's `urllib` parsing modules to do it for you.

Table of contents

- [What is a URL query string?](#)
- [Invalid characters for URLs](#)
 - [How Python's urllib handles invalid URL characters](#)
 - [Using `urllib.parse.quote` to escape invalid characters](#)
 - [Serializing dictionaries into query strings](#)
- [Requests to the rescue](#)
- [Trying out Google Static Maps](#)
 - [A review of the static maps parameters](#)
 - [Adding markers](#)
 - [Programmatically create query strings for Google Static Maps API](#)
 - [Basic Google Static Maps API URL](#)
 - [Adding zoom](#)
 - [Adding markers](#)
 - [Using `urlencode`'s `doseq` parameter to specify a list of values](#)
 - [Mapping earthquakes](#)
- [Let's make a function for mapping markers](#)
- [Styling markers](#)
 - [Creating a `create_styled_marker\(\)` function](#)

What is a URL query string?

A typical URL looks very much like a system file path, e.g.

```
http://www.example.com/index.html
```

A **query string** is a convention for appending key-value pairs to a URL. The standard URL for the New York Times's website's [New York](#) section is this:

```
http://www.nytimes.com/section/nyregion
```

However, if you click on the [New York](#) tab via the nytimes.com homepage, you'll notice that a whole bunch of characters are appended to the URL:

```
http://www.nytimes.com/section/nyregion?action=click&pgtype=Home
```

The **question mark** `?` denotes the separation between the standard URL and the query string. Everything after that is a key value pair, with each pair separated by an **ampersand**, `&`. The **equals sign** `=` is used to separate key and value.

So the key-value pairs in the above query string are:

| key | value |
|--------|----------|
| action | click |
| pgtype | Homepage |

Or, more to our purposes, this is what those keypairs would look like as a **dictionary**:

```
{
  'action': 'click',
  'pgtype': 'Homepage'
}
```

What do those actually *do*? That's actually a question we can't answer, unless we're running the nytimes.com servers. Though it's safe to assume that the NYT uses the query string in its analytics, so that it can tell how many people visited `http://www.nytimes.com/section/nyregion` via the homepage and by clicking on some button.

Other web services have query strings that serve a more obvious purpose. For example, **DuckDuckGo**, which has this URL endpoint:

```
https://www.duckduckgo.com/
```

However, if we append a key-pair value as a query string, with `q` being the **key** (think of it as an abbreviation for "*query*") and the **value** being the term we want to search for, e.g. `Stanford`, then DuckDuckGo will return [search results for Stanford](#):

```
https://www.duckduckgo.com/?q=Stanford
```

Invalid characters for URLs

It's hard to tell these days because modern web browsers allow us to literally type in anything from the keyboard. Unless we prepend the input with `http://`, the text is just sent as is to Google or DuckDuckGo or whatever your default search engine is.

However, this is just a convenient illusion. When we type into your browser bars, say, something with a whitespace character, e.g.

Stanford University

Before sending it to the search engine, the web browser will actually *serialize* it as:

Stanford%20University

This is because whitespace characters are **not allowed** in URLs, so the token `%20` is used to represent it. Basically, almost everything that is not an [alphanumeric character needs to have this special encoding](#).

In the olden days, you'd have to remember how to do these encodings or else the browser would throw you an error. Now, the browser just fixes it for you, not unlike an auto spellchecker.

How Python's urllib handles invalid URL characters

Of course, when programming in Python, *things still work like the olden days* – i.e. we're forced to be `_explicit`.

Here's what happens when you use the `urllib.retrieve` method that comes via Python's built-in module `urllib.request`:

```
from urllib.request import urllib2
thing = urllib2.urlopen("https://www.duckduckgo.com/?q=Stanford Univ
```

An error is raised:

```
HTTPError: HTTP Error 400: Bad Request
```

We have to throw in the `%20` ourselves to avoid the error:

```
thing = urllib2.urlopen("https://www.duckduckgo.com/?q=Stanford%20Univ
```

Using `urllib.parse.quote` to escape invalid characters

Trying to remember which characters are invalid, nevermind manually escaping them with percent signs, is a maddening task. That's why there's a built-in Python module – [urllib.parse](#) – that contains an appropriate method: `quote`.

Try it out via interactive Python – note that `parse` doesn't actually do any URL requesting itself – it is a method that does one thing and one thing well: making strings safe for URLs:

```
>>> from urllib.parse import quote
>>> quote("Stanford University")
'Stanford%20University'
>>> quote("I go to: Stanford University, California!")
'I%20go%20to%3A%20Stanford%20University%2C%20California%21'
```

In combination with the previously-tried `urlretrieve` method:

```
from urllib.request import urlretrieve
from urllib.parse import quote
qstr = quote("Stanford University")
thing = urlretrieve("https://www.duckduckgo.com/?q=" + qstr)
```

Serializing dictionaries into query strings

In the previous example, having to type out that `q=` should *also* seem tedious to you. Once again, [urllib.parse](#) has a method for that: `urlencode`.

Try it out in interactive Python:

```
>>> from urllib.parse import urlencode
>>> mydict = {'q': 'Stanford University, whee!!!'}
>>> urlencode(mydict)
'q=Stanford+University%2C+whee%21%21%21'
```

Note that the `urlencode` method includes the functionality of the `quote` function, so you probably rarely need to call `quote` on its own. Also note that `urlencode` uses the `plus` sign to encode a space character in a URL...which is basically as valid as using `%20`. Again, the confusing rules and standards are yet another reason to delegate this string parsing to the proper Python libraries.

And, again, `urlencode` does not actually fetch the URL. We still have to use `urlretrieve`:

```
from urllib.request import urlretrieve
from urllib.parse import urlencode
mydict = {'q': 'whee! Stanford!!!', 'something': 'else'}
qstr = urlencode(mydict)
# str resolves to: 'q=whee%21+Stanford%21%21%21&something=else'
thing = urlretrieve("https://www.duckduckgo.com/?" + qstr)
```

Note that we also have to include the `?`, which is always used to set off the query string from the first part of the URL.

Also note that programatically fetching search queries via DuckDuckGo (or Google, for that matter)...is not very effective. I just use it as an example so that you can see what the URL turns out to be and test it in your browser.

Requests to the rescue

What about the [Requests library](#), which we've been using to fetch URLs for the most part? Well, true to its slogan of being "HTTP for Humans", the Requests library neatly wraps up all that `urllib.parse` functionality for us.

Just use the `requests.get` method with a second argument (the name of the argument is `params`):

```
import requests
url_endpoint = 'https://www.duckduckgo.com'
mydict = {'q': 'whee! Stanford!!!', 'something': 'else'}
resp = requests.get(url_endpoint, params=mydict)
```

Trying out Google Static Maps

Let's work with a more fun, visual API: [the Google Static Maps API](#)

(For more information on Google Static Maps API, check out: [Visualizing Geopolitical Sensitivities with the Google Static Maps API](#))

A review of the static maps parameters

As with most APIs, Google Static Maps starts out with a URL endpoint:

```
https://maps.googleapis.com/maps/api/staticmap
```

At a minimum, it requires a **size** parameter, with a value in the format of **WIDTHxHEIGHT**:

```
https://maps.googleapis.com/maps/api/staticmap?size=600x400
```

Here's what that map looks like:




Let's add another parameter: **zoom**

```
https://maps.googleapis.com/maps/api/staticmap?size=600x400&zoom=8
```


And let's change where the map is centered around with the **center** parameter, which takes any string that describes a human-readable location:

```
https://maps.googleapis.com/maps/api/staticmap?size=600x400&zoom=8&center=Chicago
```

 `https://maps.googleapis.com/maps/api/staticmap?size=600x400&zoom=8¢er=Chicago`

Or, we can pass in a latitude/longitude pair:


```
https://maps.googleapis.com/maps/api/staticmap?size=600x400&zoom=8&center=42,-70
```

 <https://maps.googleapis.com/maps/api/staticmap?size=600x400&zoom=8¢er=42,-70>

Adding markers

Maybe you're thinking: *this is easy, handcoding the URL parameters*. Let's do something tricky. The **markers** parameter lets us add markers to the map. It takes a location string (just like **center**):

```
https://maps.googleapis.com/maps/api/staticmap?size=600x400&markers=Stanford,CA
```


 <https://maps.googleapis.com/maps/api/staticmap?size=600x400&markers=Stanford,CA>

However, the API allows for the marking of multiple points (hence, the parameter plural name of "markers"). The standard for URL query strings when multiple values have the same key is to *repeat* the key. In other words, to show **markers** for both **Stanford, CA** and **Chicago**, we include this in the query string:

```
markers=Stanford,CA&markers=Chicago
```

e.g.

```
https://maps.googleapis.com/maps/api/staticmap?size=600x400&markers=Stanford,CA&markers=Chicago
```

 <https://maps.googleapis.com/maps/api/staticmap?size=600x400&markers=Stanford,CA&markers=Chicago>

Customizing the style of markers

Still seem simple? Let's try to add different colors. And [icons](#). According to the [documentation on markers](#), styling a marker involves setting "a series of value assignments separated by the pipe (

Because both style information and location information is delimited via the pipe character, style information must appear first in any marker descriptor. Once the Google Static Maps API server encounters a location in the marker descriptor, all other marker parameters are assumed to be locations as well.

By default, the map markers are **red**. To make a marker **green**, this is what the **markers** value is set to:

```
markers=color:green|Chicago
```

e.g.

```
https://maps.googleapis.com/maps/api/staticmap?size=600x400&mark
```

| | | |
|--|--|--------|
| ! | Chicago] | Chicag |
| [https://maps.googleapis.com/maps/api/staticmap? | (https://maps.googleapis.com/maps/api/staticmap? | |
| size=600x400&markers=color:green | size=600x400&markers=color:green | |

We can also change the icon's size. And give it a letter. Here's the value for a blue icon for Chicago, with a label consisting of the letter "X":

```
markers=color:blue|label:X|Chicago
```

| | |
|--|--|
| ! | label:X Chicago] |
| [https://maps.googleapis.com/maps/api/staticmap? | (https://maps.googleapis.com/maps/api/staticmap? |
| size=600x400&markers=color:blue | size=600x400&markers=color:blue |

Technically, while the URL examples above will work in a modern browser, pipe characters aren't allowed in URLs. They should be escaped with **%7C** :

```
markers=color:blue%7Clabel:X%7CChicago
```

If that's not convoluted/ugly enough for you, the Google Maps API lets you use [custom icons](#). Here's how to make a marker using President Obama's face (found at the [following URL](#):

```
http://www.compciv.org/files/images/randoms/obamaicon.png
```

The corresponding **markers** value:

```
markers=icon:http://www.compciv.org/files/images/randoms/obamaicon
```

However, as you can imagine, some of those characters in the URL are *not* allowed as is in the query string. Think about it; we're putting a URL inside another URL...how would a primitive browser easily parse that? The URL example above is pretty simple, but since URLs can contain any number of strange characters, we should assume many of those characters will have to be encoded in that percent-fashion. From the [Google API's explanation](#):

Note: The multiple levels of escaping above may be confusing. The Google Chart API uses the pipe character (`|`) to delimit strings within its URL parameters. Since this character is not legal within a URL (see the note above), when creating a fully valid chart URL it is escaped to %7C. Now the result is embedded as a string in an icon specification, but it contains a % character (from the %7C mentioned above), which cannot be included directly as data in a URL and must be escaped to %25. The result is that the URL contains %257C, which represents two layers of encoding. Similarly, the chart URL contains an & character, which cannot be included directly without being confused as a separator for Google Static Maps API URL parameters, so it too must be encoded.

This is what the URL ends up being:

```
https://maps.googleapis.com/maps/api/staticmap?size=600x400&mark
```

And here's Obama's head, floating over Chicago:



Programmatically create query strings for Google Static Maps API

OK, now that we've seen how to do things the painful and old-fashioned way, let's use the `urllib.parse.urlencode` method to do the painful work of creating the URL query strings.

First, let's import `urlencode` and set up the constants, i.e. the variables that *won't* change:

```
from urllib.parse import urlencode
import webbrowser
GMAPS_URL = 'https://maps.googleapis.com/maps/api/staticmap?'
```

Importing `webbrowser` is optional. But it allows you to conveniently test out the URLs from your Python code:

```
import webbrowser
url = "https://maps.googleapis.com/maps/api/staticmap?size=600x400&mark=1"
webbrowser.open(url)
```

(You should be doing this in interactive Python)

Here we go.

Basic Google Static Maps API URL

Only the `size` parameter needs to be specified:

```
mydict = {'size': '600x400'}
url = GMAPS_URL + urlencode(mydict)
# https://maps.googleapis.com/maps/api/staticmap?size=600x400
```

Adding zoom

This is simply another key-value pair in the dictionary:

```
mydict = {'size': '600x400', 'zoom': 4}
url = GMAPS_URL + urlencode(mydict)
# https://maps.googleapis.com/maps/api/staticmap?size=600x400&zoom=4
```

Adding center

Again, just another key-value pair


```
mydict = {'size': '600x400', 'zoom': 9, 'center': 'Stanford, Cal
url = GMAPS_URL + urlencode(mydict)
# https://maps.googleapis.com/maps/api/staticmap?center=Stanford
```

Adding markers

`markers` is just another key-value pair, when we're just adding a single marker:

(note that I've removed the `center` param...the Google Static Maps API is smart enough to just auto-center the map around the marker)

```
mydict = {'size': '600x400', 'zoom': 14,
          'markers': 'CoHo Café at Stanford'}
url = GMAPS_URL + urlencode(mydict)
# https://maps.googleapis.com/maps/api/staticmap?markers=CoHo+Ca
```

Using urlencode's doseq parameter to specify a list of values

OK, adding multiple markers is where things get slightly complicated. We can represent a list of location strings by using, well, a list of strings:

```
locations = ['Stanford, CA', 'Berkeley, CA']
```

However, we must call `urlencode` with the `doseq` argument set to `True` (try omitting the argument to see what results on your own):

```
locations = ['Stanford, CA', 'Berkeley, CA']
mydict = {'size': '600x400', 'markers': locations}
url = GMAPS_URL + urlencode(mydict, doseq=True)
# https://maps.googleapis.com/maps/api/staticmap?markers=Stanford
```

Just for fun, serialize a bunch of locations into a Google Static Maps URL:

```
pac12 = ['University of Arizona, AZ',
         'Arizona State University, AZ',
         'University of California, Berkeley, CA',
         'University of California, Los Angeles, CA',
         'University of Colorado Boulder, CO',
         'University of Oregon, OR',
         'Oregon State University, OR',
         'University of Southern California, CA',
         'Stanford University, CA',
         'University of Utah, UT',
         'University of Washington, WA',
         'Washington State University, WA']

mydict = {
    'size': '600x400', 'markers': pac12
}

url = GMAPS_URL + urlencode(mydict, doseq=True)
```

The resulting image and URL:



Mapping earthquakes

It's not a lot of fun creating lists by hand. So let's use a list from an official government source: [the USGS Earthquake Hazards program](#).

The example below is a demonstration of the `csv` module and the `DictReader` function, which can be used to create a list of dictionaries from a CSV file.

```
import csv
import requests
from urllib.parse import urlencode
import webbrowser

USGS_URL = 'http://earthquake.usgs.gov/earthquakes/feed/v1.0/summary'
resp = requests.get(USGS_URL)
lines = resp.text.splitlines()
earthquakes = csv.DictReader(lines)
coordinate_pairs = []
for quake in earthquakes:
    coordinate_pairs.append(quake['latitude'] + ',' + quake['longitude'])
# create a URL based on Google Static Maps API specs
GMAPS_URL = 'https://maps.googleapis.com/maps/api/staticmap'
query_string = urlencode(
    {'size': '800x500', 'markers': coordinate_pairs})
url = GMAPS_URL + '?' + query_string
webbrowser.open(url)
```

Here's the resulting URL (as of February 9th, 2016):

<https://maps.googleapis.com/maps/api/staticmap?size=800x500&markers=-6.6214%2C154.7073&markers=22.9387%2C120.5928&markers=54.007%2C158.5674.3&markers=35.6519%2C-3.6755&markers=59.6047%2C-153.3405&markers=18.8414%2C-106.9536&markers=41.9519%2C142.7205&markers=3.8769%2C126.8708>

Here's a slightly more cleaned-up version of the code that uses the Requests library to "prepare" a URL:

```
# slightly more Pythonic, cleaner version
from csv import DictReader
import requests
import webbrowser

USGS_URL = 'http://earthquake.usgs.gov/earthquakes/feed/v1.0/sum'
GMAPS_URL = 'https://maps.googleapis.com/maps/api/staticmap'

# get the USGS data, create a list of lines
lines = requests.get(USGS_URL).text.splitlines()

# get the latitude/longitude pairs
coordinate_pairs = ["%s,%s" % (q['latitude'], q['longitude']) fo

# this is another way of serializing the URL
preq = requests.PreparedRequest()
preq.prepare_url(GMAPS_URL, {'size': '800x500', 'markers': coordi
webbrowser.open(preq.url)
```

Let's make a function for mapping markers

Before we get into the tricky business of *styling* the markers, let's wrap up the functionality that we've been using to create a proper Google Static Maps URL into a function:

(assuming you've read the short guide on making functions: [Function fundamentals in Python](#))

Here's a bare-bones implementation, in which a user only has to specify a **list** (or just a string, if there's only one location) of locations and, optionally, width and height. The `foo_goo_url` function does the work of serializing the input into proper Google Static Maps API format.

At the end, it returns the URL as a string object:

```
def foogoo_url(locations, width = 600, height = 400):
    from urllib.parse import urlencode
    gmap_url = 'https://maps.googleapis.com/maps/api/staticmap'
    size_str = str(width) + 'x' + str(height)
    query_str = urlencode({'size': size_str, 'markers': location
    return gmap_url + '?' + query_str
```

Here's how you would use it, interactively:

```
>>> foogoo_url('New York, NY', height='200')
'https://maps.googleapis.com/maps/api/staticmap?markers=New+York'
>>> foogoo_url(['Wyoming', 'Alaska'])
'https://maps.googleapis.com/maps/api/staticmap?markers=Wyoming&
```

Testing out those URLs by pasting them into the web browser is so time-consuming. So let's make another function. This one doesn't return anything. Instead, it takes the same parameters as `foogoo_url`, but passes them directly into `foogoo_url`, and then passes the result of that into `webbrowser.open`, which performs the action of opening a webbrowser:

(note that this definition assumes that `foogoo_url` has been defined earlier)

```
def foogoo_browse(locations, width=600, height=400):
    import webbrowser
    url = foogoo_url(locations, width, height)
    webbrowser.open(url)
```

Styling markers

(Note: This section ends up veering out of plain URL creation and into application and function design...I end up not quite finishing it...)

This is where it gets tricky. As a reminder, `markers` takes a pipe-delimited string to separate the style configuration, e.g.

```
markers=color:green|label:Z|Chicago
```

However, that's a convention of Google's own making, because they needed a way to do key-value pairs (e.g. `label = Z`) that is independent (or rather, *nested*) in the way that key-value pairs are done in URL query strings.

So basically, we have to hand-create the string ourselves:

```
marker_styles = {
    'color': 'orange',
    'label': 'X'
}
marker_config = []
for k, v in marker_styles.items():
    s = str(k) + ':' + str(v)
    # e.g. 'color' + ':' + 'orange'
    marker_config.append(s)
# the location always comes last, as specified by the Google API
marker_config.append('Chicago')
# now join the list elements together as a pipe-delimited string
marker_string = '|'.join(marker_config)
```

The result of the code above would result in the `marker_string` variable pointing to a string like this:

```
color%3Aorange%7Clabel%3AX%7CChicago
```

Which we can then pass into the `locations` argument of our previously defined `foogoo_url` function:

```
foogoo_url(marker_string)
```

Which generates a URL like this:

```
https://maps.googleapis.com/maps/api/staticmap?markers=color%3Aorange%7Clabel%3AX%7CChicago
```

Defining the marker style certainly got complicated...it's so complicated that it probably deserve its own method.

Creating a `create_styled_marker()` function

I'm going to skip the full explanation, or bother even creating what I consider to be the best real-world implementation of a `create_styled_marker` function. We can cover it in another lesson, but the main takeaway is: look at how we can use functions and Python data structures, such as **lists** and **dictionaries**, to create text strings useful for communicating with other services.

Without further elaboration, here's how to turn the previous icon mapping code into a reusable function:

```
def create_styled_marker(location, style_options={}):
    mconfig = []
    for k, v in style_options.items():
        mconfig.append(str(k) + ':' + str(v))
    mconfig.append(location)
    return '|'.join(mconfig)
```

Note: if lists and dictionaries are old hat to you and you understand [list comprehensions](#), as [well as string formatting](#), here's a fancy pants Python-version:

```
def create_styled_marker(location, style_options={}):
    opts = ["%s:%s" % (k, v) for k, v in style_options.items()]
    opts.append(location)
    return '|'.join(opts)
```

Here's all the relevant code to create a quickie-let's-make-a-Google-Static-Maps-API convenience wrapper, as one big script:

Note that I've drastically modified `foogoo_url` from the previous demonstration. See if you can untangle the reasoning...but it's not worth explaining in full since this isn't a lesson on application design...

```

from urllib.parse import urlencode
import webbrowser

def create_styled_marker(location, style_options={}):
    opts = ["%s:%s" % (k, v) for k, v in style_options.items()]
    opts.append(location)
    return '|'.join(opts)

def foogoo_url(locations, width = 600, height = 400, maptype='te
    gmap_url = 'https://maps.googleapis.com/maps/api/staticmap'
    size_str = str(width) + 'x' + str(height)
    # note: this is messy, and it has more to do with opinions a
    # design than the core lesson...
    if type(locations) is not list:
        markers_objects = locations
    else:
        markers_objects = []
        for loc in locations:
            if type(loc) is str:
                obj = create_styled_marker(loc)
            else:
                obj = create_styled_marker(loc[0], loc[1])
            markers_objects.append(obj)
    #finally, make the query string
    mapopts = {'size': size_str,
               'markers': markers_objects,
               'maptype': maptype}
    query_str = urlencode(mapopts, doseq=True)
    return gmap_url + '?' + query_str

def foogoo_browse(locations, width=600, height=400):
    import webbrowser
    url = foogoo_url(locations, width, height)
    webbrowser.open(url)

```

And when the functions are defined and loaded into the interpreter, this is how we call the functions:

```

foogoo_browse("Stanford University, California")
# a list of strings
foogoo_browse(['Stanford, CA', 'Chicago, IL'])
# multiple kinds of objects
mylist = []
mylist.append(['Stanford, CA', {'color': 'red'}])
mylist.append(['Berkeley, CA', {'color': 'yellow', 'size': 'small'}])
foogoo_browse(mylist)

```

References and Related Readings

[Google Static Map Maker](#)

[Query string](#) [Wikipedia]

[Answer to: "Which characters make a URL invalid?"](#) [Gumbo; Stack Overflow]

[Downloading files with the Requests library](#)

Using the Requests library for the 95% of the kinds of files that we want to download.

[urllib.parse - Parse URLs into components](#) [Python documentation]

Computational Methods in the Civic Sphere is produced by [Dan Nguyen](#) as an online reference for the [Stanford Computational Journalism Lab](#) and [Masters in Journalism](#) curriculum.

[Assignments](#)
[Articles](#)

Links

[Stanford Computational Journalism Lab](#)
[Stanford Journalism Masters Program](#)