

Geocoding with the Google Maps API: Using PROC FCMP To Call User-Defined SAS and Python Functions That Geocode Coordinates into Addresses, Calculate Routes, and More!

Troy Martin Hughes

ABSTRACT

Software interoperability describes the ability of software systems, components, and languages to communicate effectively with each other, and must be prioritized in today's multilingual and open-source development environments. PROC FCMP, the SAS® Function Compiler, enables Python functions to be wrapped in (and called from) SAS user-defined functions (and subroutines), and the full panoply of SAS environments supports FCMP—including SAS Display Manager, SAS Enterprise Guide, SAS Studio, SAS Viya, and the latest Cary show pony, SAS Viya Workbench. Productivity and the pace of development are maximized when existing open-source code, such as Python user-defined functions, can be run natively from a Python interactive development environment (IDE) rather than having to be needlessly recoded into the Base SAS language. This text demonstrates SAS and Python user-defined functions that collaboratively call the Google Maps Platform APIs to geocode street addresses into latitude/longitude coordinates, and to calculate driving distances between locations. The scenarios in this text demonstrate the application of the Google Maps Platform for clinical trials research, and the technical concepts are adapted from the renowned SAS Press book: *PROC FCMP User-Defined Functions: An Introduction to the SAS® Function Compiler, Second Edition*. (HUGHES, 2026)

INTRODUCTION

Geocoding describes the act of ascribing geolocal coordinates (e.g., latitude and longitude) to a location, typically a street address. Coordinates are often required (in lieu of an address) to perform certain analyses, such as a geospatial join of data (in which attributes, like population density or average socioeconomic status of residents, are attributed to a set of coordinates). In other cases, coordinates are required to perform a point-in-polygon geospatial analysis, through which the attributes of some bounded region (like a city, county, or state) are applied to points (locations) inside that region. For example, in the realm of clinical trials, given a subject's home address, geocoding that address would yield the corresponding latitude/longitude coordinates, which in turn could be geospatially joined to evaluate the population density, socioeconomic status, obesity rates, and other potentially indicating or contributing health factors for that region.

Consider a real-world example and patient—let's call him Q—who suffers from purulent rashes—ew! It's no fun having Q-goo, but rather than sit and merely stew, our hero Q joins the pharmacological clinical trial queue for a topical ointment to treat his Q-goo. And with any luck, in a month or two, his pus-filled skin that drips *au jus* will be restored to health and born anew.

The clinical research organization (CRO) needs as much environmental data as possible to understand any effects that Q's location might have on either the severity of his condition or the success of his treatment. Geocoding Q's street address—converting it to latitude/longitude coordinates—is required to ascribe the socioeconomic status of Q's neighborhood (retrieved from US Census data, although not demonstrated in this text) to his patient record. Geocoding is also required (directly or indirectly) to calculate the distance between Q's home and various clinics where he receives treatment—to understand whether and how subject proximity (to service location) is correlated with missed visits, adherence to treatment regimens, or treatment success. For example, a principal investigator might hypothesize that subjects willing to travel greater distances for treatment are more invested in their treatment and have better outcomes.

The Google Maps API offers unfettered access not only to Google's rich tapestry of geolocal data but also to its array of built-in operations that evaluate and transform these data. And because the API can be accessed via Python methods (among other open-source languages), it is understandable that further

development would occur in Python, and not in proprietary software like SAS. So, what to do about a SAS-loving CRO that must incorporate an API like Google Maps to ingest and analyze geolocation data?

To facilitate this software interoperability—between proprietary SAS and open-source Python—and to do so smartly and efficiently, the SAS function compiler—PROC FCMP—enables Python functions to be called from the SAS application, and yet run natively from a Python interpreter. Espousing this interoperability enables SAS practitioners to build SAS solutions and Python developers to build Python solutions, and for these systems to work cohesively while being maintained and modified independently.

INTRODUCING THE GOOGLE MAPS (DEVELOPER) PLATFORM

The Google Maps Platform is the development portal for accessing Google geospatial offerings, including Google APIs (and supporting code) that geocode addresses (into coordinates), reverse geocode coordinates (into street addresses), generate walking or driving directions, generate 2D and 3D maps, and so much more. (Google, 2025)

The screenshot shows the Google Maps Platform website. At the top is a navigation bar with links for Overview, Products, Pricing, and More. There is a search bar, a language selector set to English, and a user profile icon. Below the navigation bar are two buttons: Contact Sales and Get Started. The main hero section has a blue background with the text "Build awesome apps with Google's knowledge of the real world" and a subtext "Create real-world, real-time experiences with the latest Maps, Routes, and Places features from Google Maps Platform. Built by the Google team for developers everywhere." Below this are three buttons: Get Started, See our new pricing structure, and Visit documentation. The "Explore top topics" section features four cards, each with a map-related image, a title, and a brief description.

Google Maps Platform Overview Products ★ Pricing More ▾ Search / English T

Contact Sales Get Started

Build awesome apps with Google's knowledge of the real world

Create real-world, real-time experiences with the latest Maps, Routes, and Places features from Google Maps Platform. Built by the Google team for developers everywhere.

Get Started See our new pricing structure Visit documentation →

Explore top topics

- Get going with Navigation SDK**
Deliver seamless, customized in-app Google Maps navigation to your Android and iOS apps.
- Make an animated 3D map**
Take this codelab to learn how to add an animated 3D map with markers to your web app.
- Get answers from Places Aggregate API**
Learn about the Preview release a new API to retrieve the density of places within a geographic area for powerful analytics and recommendations.
- React components for the Maps JavaScript API**
The lead engineers behind the @vis.gl/react-google-map library walk through its technical design.

Within the overarching Google Maps Platform, numerous APIs support varied functionality, and because access to the Google Maps Platform requires a Google Cloud account (so that API usage can be billed once it surpasses established thresholds), developers are able to select only those services of interest, and can opt in or opt out of all APIs to customize functionality and ensure accurate billing. The Google Cloud account imparts a Google Maps API key, which is utilized in Python calls to track which user (or application or organization) should be billed for webservices. This key tracks calls to the API from a specific account, and if the threshold of free calls (per month) is surpassed, it facilitates billing for these API services. For this reason, it is critical that your Google Maps API key be privately and securely maintained.

The Google Maps APIs and Services page lists available APIs, including some that have been deprecated and are being transitioned to newer versions. (Google, 2025) In the following screen capture, note that Geocoding and Geolocation are optionally enabled, whereas Places UI and Places are optionally disabled.

The screenshot shows the Google Cloud console page for Google Maps Platform APIs & Services. The left sidebar contains navigation links: Overview, Metrics, Quotas, Keys & Credentials, Support, Solution Library, Map Management, Map Styles, and Datasets. The main content area displays a grid of API cards. Each card includes an icon, the API name, a brief description, and a status toggle (Enable/Disable). The APIs shown are: Places UI Kit (disabled), Geocoding API (disabled), Geolocation API (disabled), Places API (New) (enabled), Time Zone API (disabled), Address Validation API (enabled), Navigation SDK (enabled), and Directions API (disabled). Each card also features a 'Places' button and a 'Guides' link.

Each API has corresponding usage instructions as well as developer guides, such as Client Libraries for Java, Python, Go, and Node.js. Only Python libraries are discussed in this text. For example, the Google Maps Geocoding API is the landing zone for developers who need to geocode addresses into geospatial coordinates. (Google, 2025)

The screenshot shows the Google Maps Platform documentation page for the Geocoding API. The left sidebar contains navigation links: Overview, Get Started, Setup, and Developer Guides. The main content area features a map of New York City with a red pin at 277 Bedford Ave, Brooklyn, NY 11211, USA. The title 'Geocoding API' is prominently displayed. Below the title, the text reads: 'Convert addresses or Place IDs to latitude/longitude coordinates and vice-versa.' A search bar is provided with the text 'Search Maps Geocoding API docs'. The right sidebar contains links: Info, Chat, and API.

The documentation for Google Client Libraries is hosted on the Google Maps Platform, and includes information about usage, as well as links to language-specific repositories. (Google, 2025)

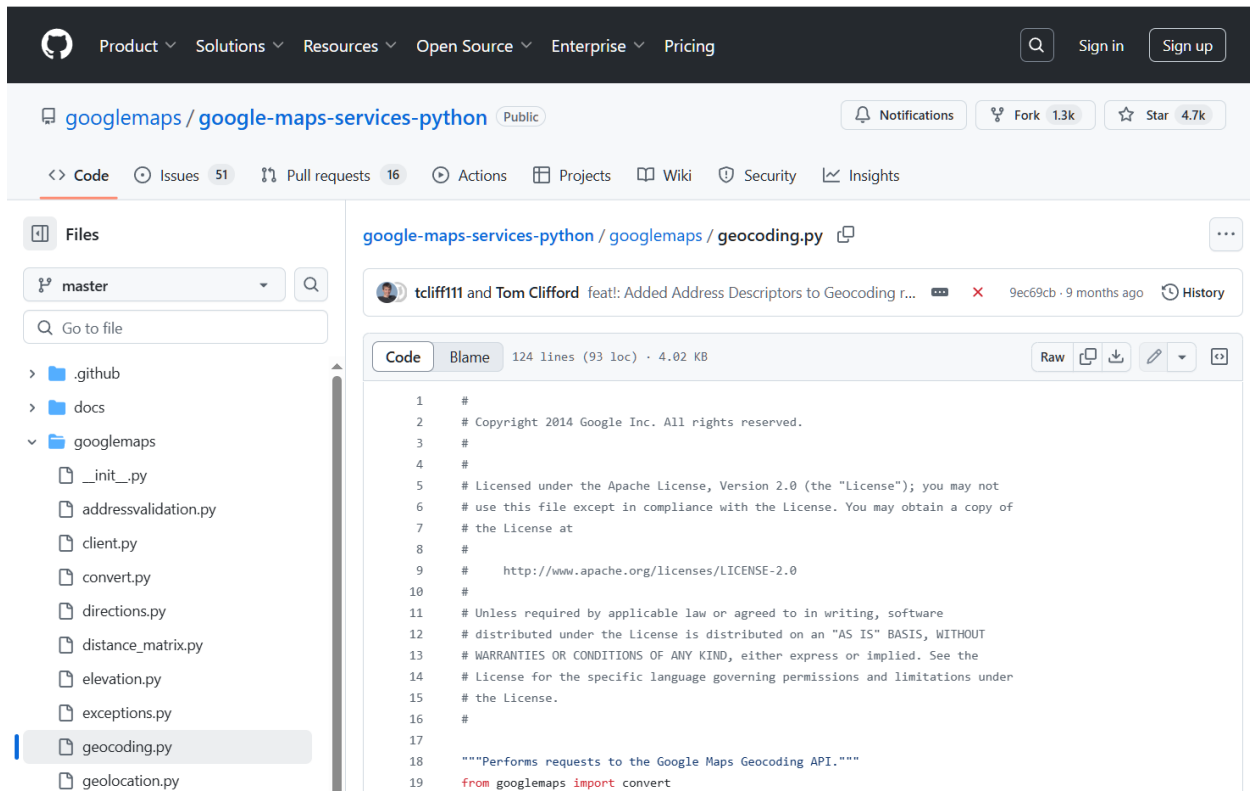
The screenshot shows the Google Maps Platform documentation page for 'Client Libraries for Google Maps Web Services'. The page is part of the 'Web Services > Geocoding API' section. It features a 'Get Started' button and a 'Contact sales' button. The main content area has a breadcrumb trail: 'Home > Products > Google Maps Platform > Documentation > Web Services > Geocoding API > Resources'. The title 'Client Libraries for Google Maps Web Services' is prominently displayed, followed by a 'Send feedback' button. Below the title, a paragraph states: 'The Java Client, Python Client, Go Client and Node.js Client for Google Maps Services are community supported client libraries, open sourced under the [Apache 2.0 License](#). They are available for download and contributions on GitHub, where you will also find installation instructions and sample code:'. On the left sidebar, there are links for 'Support', 'Support Options', 'Maps FAQ', 'Geocoding FAQ', 'Release Notes', 'Stay Informed', 'Terms of Use', 'Best Practices', and 'Web Service Best Practices'. On the right sidebar, there are links for 'Info', 'Chat', and 'API'.

For example, from this Client Libraries page, the google-maps-services-python GitHub repository can be accessed, in which the API methods and other code are documented and maintained. (GitHub, 2025)

The screenshot shows the GitHub repository page for 'googlemaps/google-maps-services-python'. The repository is public and has 1.3k forks and 4.7k stars. The page shows the repository structure on the left, including files like '.__init__.py', 'addressvalidation.py', 'client.py', 'convert.py', 'directions.py', 'distance_matrix.py', 'elevation.py', 'exceptions.py', 'geocoding.py', 'geolocation.py', 'maps.py', 'places.py', 'roads.py', 'timezone.py', and a 'tests' directory. The main content area shows a list of commits. The most recent commit is by 'tcliff111 and Tom Clifford' with the message 'feat!: Added Address Descriptors to Geocoding r...' and a date of '9 months ago'. Below this, a table lists the commit history with columns for 'Name', 'Last commit message', and 'Last commit date'.

Name	Last commit message	Last commit date
..		
__init__.py	chore(release): 4.10.0 [skip ci]	2 years ago
addressvalidation.py	fix: fixes broken support for python 3.5 (#453)	3 years ago
client.py	fix: correct lazy attribute inside logger (#457)	3 years ago
convert.py	fix: fixed code quality issues (#398)	4 years ago
directions.py	docs: fix simple typo, prefix -> prefix (#389)	4 years ago
distance_matrix.py	docs: specify support for place ID parameter in distan...	4 years ago
elevation.py	Return empty result on APIs when key missing from re...	8 years ago
exceptions.py	fix: APIError.__str__ should always return a str (#328)	6 years ago
geocoding.py	feat!: Added Address Descriptors to Geocoding respo...	9 months ago
geolocation.py	Make it subclass ApiError	8 years ago
maps.py	docs: fix code format in static maps docs (#475)	2 years ago
places.py	feat: add new place details fields and reviews request ...	2 years ago
roads.py	Make it subclass ApiError	8 years ago
timezone.py	Support POST requests in client.	9 years ago

And finally, the `geocoding.py` source file can be viewed to further understand the webservices provided by the Geocoding API. (GitHub, 2025)



The screenshot shows the GitHub interface for the repository `googlemaps/google-maps-services-python`. The file `geocoding.py` is selected in the file explorer on the left. The main content area displays the code for `geocoding.py`, which includes a license header and a function for geocoding. The code is as follows:

```
1 #
2 # Copyright 2014 Google Inc. All rights reserved.
3 #
4 #
5 # Licensed under the Apache License, Version 2.0 (the "License"); you may not
6 # use this file except in compliance with the License. You may obtain a copy of
7 # the license at
8 #
9 # http://www.apache.org/licenses/LICENSE-2.0
10 #
11 # Unless required by applicable law or agreed to in writing, software
12 # distributed under the license is distributed on an "AS IS" BASIS, WITHOUT
13 # WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
14 # License for the specific language governing permissions and limitations under
15 # the license.
16 #
17
18 """Performs requests to the Google Maps Geocoding API."""
19 from googlemaps import convert
```

With this introduction to the Google Maps Platform, the remainder of this text leverages Python methods within the Google Maps libraries to geocode street addresses and to calculate the distance between two locations. In all instances, only Python methods from the Google Maps API call Google webservices—and user-defined SAS functions and subroutines call the user-defined Python functions that leverage those Python methods. That is, FCMP empowers SAS practitioners to construct multiple levels of abstraction that facilitate calling open-source code from the SAS DATA step.

GEOCODING A LOCATION USING GOOGLE MAPS

To protect Q's home address, let's say that Q, our hapless subject, lives in the Navy Lodge on Naval Base San Diego. In only a few lines of Python code, his address is geocoded into coordinates using the `geocode` method (after first passing the Google Maps API key):

```
# import the Google Maps package
import googlemaps

# initialize the Google Maps key
gmaps_key = googlemaps.Client(key="YOUR_SECRET_KEY_GOES_HERE")

# initialize subject address to Navy Lodge, Naval Base San Diego
pt_addy = 'Naval Base San Diego Bldg #3526, San Diego, CA 92136'

# geocode the address
geo = gmaps_key.geocode(pt_addy)
```

The data structure for `geo`, the resultant geocoded output, is demonstrated:

```
[{'address_components': [{'long_name': '3455',
  'short_name': '3455',
  'types': ['street_number']},
  {'long_name': 'Senn Street', 'short_name': 'Senn St', 'types': ['route']}],
```

```
{'long_name': 'Barrio Logan',
 'short_name': 'Barrio Logan',
 'types': ['neighborhood', 'political']},
{'long_name': 'San Diego',
 'short_name': 'San Diego',
 'types': ['locality', 'political']},
{'long_name': 'San Diego County',
 'short_name': 'San Diego County',
 'types': ['administrative_area_level_2', 'political']},
{'long_name': 'California',
 'short_name': 'CA',
 'types': ['administrative_area_level_1', 'political']},
{'long_name': 'United States',
 'short_name': 'US',
 'types': ['country', 'political']},
{'long_name': '92136', 'short_name': '92136', 'types': ['postal_code']}},
'formatted_address': '3455 Senn St, San Diego, CA 92136, USA',
'geometry': {'location': {'lat': 32.6848432, 'lng': -117.1300151},
 'location_type': 'ROOFTOP',
 'viewport': {'northeast': {'lat': 32.6861158302915,
 'lng': -117.1287738197085},
 'southwest': {'lat': 32.6834178697085, 'lng': -117.1314717802915}}},
'navigation_points': [{'location': {'latitude': 32.684517,
 'longitude': -117.130137},
 'restricted_travel_modes': ['WALK']},
 {'location': {'latitude': 32.6849949, 'longitude': -117.13018},
 'restricted_travel_modes': ['DRIVE']}],
'partial_match': True,
'place_id': 'ChIJ402zjVZS2YARgJhRq5k14dI',
'plus_code': {'compound_code': 'MVM9+WX San Diego, CA',
 'global_code': '8544MVM9+WX'},
'types': ['establishment', 'point_of_interest']]}
```

To elucidate this data structure, the following code demonstrates the keys of the resultant dictionary:

```
print(list(geo[0]))

['address_components', 'formatted_address', 'geometry', 'navigation_points',
'partial_match', 'place_id', 'plus_code', 'types']
```

Latitude and longitude coordinates are accessed via the **geometry** key:

```
print(geo[0]['geometry'])

{'location': {'lat': 32.6848432, 'lng': -117.1300151}, 'location_type': 'ROOFTOP',
'viewport': {'northeast': {'lat': 32.6861158302915, 'lng': -117.1287738197085},
'southwest': {'lat': 32.6834178697085, 'lng': -117.1314717802915}}}
```

The **location** key contains key-value pairs for latitude and longitude (**lat** and **lng**), and the following code extracts these values:

```
lat = geo[0]['geometry']['location']['lat']
lon = geo[0]['geometry']['location']['lng']
```

Thus, a user-defined function can be built that extracts only latitude and longitude values from a geolocated address, and returns these values as a tuple:

```
# function evaluates street address and extracts lat/lon coords
def get_coords(addr):
    geo = gmaps_key.geocode(addr)
    lat = geo[0]['geometry']['location']['lat']
    lon = geo[0]['geometry']['location']['lng']
    return lat, lon
```

When **get_coords** is called, **pt_lat** and **pt_lon** are initialized, respectively, to the latitude and longitude values returned (in a tuple) by the function:

```
# geocode address
pt_lat, pt_lon = get_coords(pt_addy)

print(pt_lat, pt_lon)

32.6848432 -117.1300151
```

But if the CRO maintains its subjects' addresses inside a SAS data set, how do these data reach the nifty **get_coords** user-defined Python function? In other words, for any development or analytic shop relying solely or primarily on SAS software, how are open-source methods, functions, and other code incorporated into SAS software, and how are data maintained so they can be transferred between (and consumed by) not only SAS but also other applications? As demonstrated in the next section, the elegance of the FCMP procedure facilitates this software interoperability and data transferability.

CALLING A PYTHON FUNCTION USING PROC FCMP

The FCMP procedure supports the Python Component Object, a SAS construct that enables Python functions to be imported into (or imbedded inside) a SAS user-defined function. (SAS Institute, 2025) To use the Python Component Object, first consult relevant SAS documentation to initialize the **MAS_M2PATH** and **MAS_PYPATH** environment variables. (SAS Institute, 2025)

After the system environment variables have been modified, one additional line of code must be added to the Python program file (saved as **get_coords.py**) that contains the user-defined **get_coords** function:

```
# import the Google Maps package
import googlemaps

# initialize the Google Maps key
gmaps_key = googlemaps.Client(key="YOUR_SECRET_KEY_GOES_HERE")

# function evaluates street address and extracts lat/lon coords
def get_coords(addy):
    "Output: lat_key, lon_key"
    geo = gmaps_key.geocode(addy)
    lat = geo[0]['geometry']['location']['lat']
    lon = geo[0]['geometry']['location']['lng']
    return lat, lon
```

Thus, the **Output** statement conveys the one or more keys that will be returned from Python (to SAS) when the **get_coords** function executes. The **lat** return value is mapped to **lat_key** and the **lon** return value is mapped to **lon_key**. With this modification to a single line of code, the **get_coords** function can now be called from the FCMP procedure.

Simulated subject data are maintained in the Subjects data set, and the following DATA step creates Q's record, which includes his street address (anonymized to the Navy Lodge):

```
* sample subject data set with addresses;
data subjects;
    length name $20 pt_addy $100 lat lon 8;
    name = 'Q';
    pt_addy = 'Naval Base San Diego Bldg #3526, San Diego, CA 92136';
run;
```

Subsequently, the **sas_get_coords** subroutine is defined (and compiled) using the FCMP procedure:

```
* define the folder location of Python program files;
%let loc_file=c:\goo\

* call get_coords Python function;
proc fcmp outlib = work.funcs.py;
    subroutine sas_get_coords(sas_addy $, sas_lat, sas_lon);
        outargs sas_lat, sas_lon;
        declare object py(python);
        rc = py.infile("&loc_file.get_coords.py");
```

```

rc = py.publish();
rc = py.call("get_coords", sas_addy);
sas_lat = py.results["lat_key"];
sas_lon = py.results["lon_key"];
endsub;
quit;

```

A user-defined subroutine is declared using the SUBROUTINE statement and is terminated with the ENDSUB statement (or the ENDFUNC statement, as ENDFUNC and ENDSUB are aliases). SAS subroutines differ from SAS functions in that subroutines do not return a value and must be called using the CALL statement, whereas functions always return a value and are called directly (without CALL). Although subroutines cannot *return* a value (because the RETURN statement is allowed only inside function definitions), both functions and subroutines can leverage the (optional) OUTARGS statement to modify one or more arguments in the calling program (through *call by reference*). Arguments not specified by OUTARGS are, by default, *called by value*.

The INFILE method specifies the Python program file in which the Python function is defined.

The PUBLISH method submits Python code to the Python interpreter.

The CALL method executes the Python function, so it passes the **sas_addy** variable to **get_coords**. At this point, **get_coords** executes, and returns a tuple containing the keys defined therein.

The RESULTS method extracts the latitude value from **lat_key** (and initializes the **sas_lat** variable), and extracts the longitude value from **lon_key** (and initializes the **sas_lon** variable).

Finally, because both **sas_lat** and **sas_lon** are specified in the OUTARGS statement, when the **sas_get_coords** subroutine terminates, the latitude and longitude values are available in the *calling program*—that is, in the DATA step that called the subroutine.

The following DATA step calls the **sas_get_coords** subroutine and initializes the **lat** and **lon** variables to the latitude and longitude, respectively, of Q's street address, maintained in the **pt_addy** variable:

```

options cmplib=work.funcs;

data analyze_pts;
  set pts;
  call sas_get_coords(pt_addy, lat, lon);
  put lat= lon=;
run;

```

Note that the OPTIONS CMPLIB statement is required and must specify the two-level name (i.e., library and data set) in which the subroutine is compiled (as previously specified by the OUTLIB option). Thus, although OUTLIB denotes **work.funcs.py** (and must include the **py** package), OPTIONS only specifies **work.funcs**.

When the DATA step executes, the latitude and longitude values are generated—geocoded from the street address—and printed to the log:

```

lat=32.6848432 lon=-117.130015
NOTE: There were 1 observations read from the data set WORK.PTS.
NOTE: The data set WORK.ANALYZE_PTS has 1 observations and 4 variables.

```

With this new functionality, the CRO can now begin to analyze its subjects with regard to their locations. And importantly, the user-defined geocoding function **get_coords** did not need to be recoded in SAS, and could run natively in Python by adding only one line of code (the **Output** statement) to the Python function's definition. This flexibility empowers SAS practitioners to maintain clinical data and analyses in SAS while Python developers separately maintain the geocoding function that interacts with the Google Maps API—or, in today's multilingual world, savvy developers adroitly maneuver in both environments.

CALCULATING ROUTE DISTANCE USING GOOGLE MAPS

At this point, the CRO is able to evaluate the geospatial coordinates of the addresses of its study subjects, which it can use in further analyses or to obtain additional information about the subjects or their respective environments. For example, as mentioned previously, a principal investigator at the CRO is interested in analyzing client location with respect to various treatment locations to understand whether “distance from treatment” correlates with illness severity or treatment outcomes.

For example, consider that the CRO has partnered with various In-N-Out Burger establishments to offer treatments for the clinical trial in its parking lots. That is, Q can drive to one of three participating San Diego restaurants to receive his weekly regimen of the topical ointment:

- 1900 E Plaza Blvd, National City, CA 91950
- 2910 Damon Ave, San Diego, CA 92109
- 10370 Friars Rd, San Diego, CA 92120

As the CRO is primarily a SAS shop, treatment center data are maintained in the `treatment_centers` data set:

```
data treatment_centers;
  length center_name $50 center_addy $100;
  call missing(lat, lon);
  center_name = 'National City In-N-Out';
  center_addy = '1900 E Plaza Blvd, National City, CA 91950';
  output;
  center_name = 'Pacific Beach In-N-Out';
  center_addy = '2910 Damon Ave, San Diego, CA 92109';
  output;
  center_name = 'Mission Valley In-N-Out';
  center_addy = '10370 Friars Rd, San Diego, CA 92120';
  output;
run;
```

To calculate routes (or route distance) programmatically using Google Maps, the origin and destination addresses (or their corresponding geospatial coordinates) are required, so although end users of Google Maps are accustomed to navigating on their phone using only street addresses, the API also accepts latitude and longitude (or Google Place IDs, not discussed in this text). Where a street address is provided to the Python method, Google first automatically geocodes this address into coordinates. Thus, directly or indirectly, geocoding is required for navigation and distance calculation.

To geocode the In-N-Out street addresses, the `sas_get_coords` subroutine (and Python `get_coords` function) can be called again:

```
data treatment_centers_coords;
  set treatment_centers;
  length lat lon 8;
  call missing(lat, lon);
  call sas_get_coords(center_addy, lat, lon);
  put lat= lon=;
run;
```

The log demonstrates that `sas_get_coords`, assisted by the Google Maps API, successfully geocoded the addresses for the three delicious treatment centers:

```
lat=32.6763009 lon=-117.085868
lat=32.8089241 lon=-117.2189284
lat=32.7909035 lon=-117.1001831
NOTE: There were 3 observations read from the data set WORK.TREATMENT_CENTERS.
```

It is not necessary to first recompile the `sas_get_coords` subroutine (by running the FCMP procedure) or to reuse the `OPTIONS CMBLIB` statement, as long as the current SAS session is maintained. However, both steps are required if SAS has been terminated and the subroutine needs to be reused in a new session.

And with this feat, the first instance of software reuse has been achieved, demonstrating one of the many ways that user-defined functions and subroutines undeniably improve the quality of software. That is, the `sas_get_coords` subroutine, as well as the underlying `get_coords` Python function that it called, were designed to geocode subject addresses, but were later *reused* to geocode the addresses for treatment centers!

Whereas the Google Maps Geocoding API geocodes street addresses into latitude/longitude coordinates, the Google Maps Distance_Matrix API (i.e, `distance_matrix` Python method) calculates distances between two locations—with the origin and destination entered either as street addresses or latitude/longitude coordinates. The API is described in the Google Maps GitHub Repository. (GitHub, 2025)

The screenshot shows the GitHub repository for `googlemaps/google-maps-services-python`. The file `distance_matrix.py` is selected in the file explorer on the left. The main content area displays the code for `distance_matrix.py`, which includes a license header and the `distance_matrix` function definition. The function takes a client, origins, and destinations as arguments and returns a list of travel distance and time for a matrix of origins and destinations.

```

1  #
2  # Copyright 2014 Google Inc. All rights reserved.
3  #
4  #
5  # Licensed under the Apache License, Version 2.0 (the "License"); you may not
6  # use this file except in compliance with the License. You may obtain a copy of
7  # the license at
8  #
9  # http://www.apache.org/licenses/LICENSE-2.0
10 #
11 # Unless required by applicable law or agreed to in writing, software
12 # distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
13 # WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
14 # License for the specific language governing permissions and limitations under
15 # the license.
16 #
17
18 """Performs requests to the Google Maps Distance Matrix API."""
19
20 from googlemaps import convert
21
22
23 def distance_matrix(client, origins, destinations,
24                     mode=None, language=None, avoid=None, units=None,
25                     departure_time=None, arrival_time=None, transit_mode=None,
26                     transit_routing_preference=None, traffic_model=None, region=None):
27     """ Gets travel distance and time for a matrix of origins and destinations.
28
29     :param origins: One or more addresses, Place IDs, and/or latitude/longitude
30                     values, from which to calculate distance and time. Each Place ID string
31                     must be prepended with 'place_id:'. If you pass an address as a string,
32                     the service will geocode the string and convert it to a
33                     latitude/longitude coordinate to calculate directions.
34     :type origins: a single location, or a list of locations, where a
35                     location is a string, dict, list, or tuple

```

For example, to calculate the distance between Q's anonymized "residence" at the Navy Lodge (Naval Base San Diego Bldg #3526, San Diego, CA 92136) and the National City In-N-Out "treatment center" (1900 E Plaza Blvd, National City, CA 91950), either the street addresses or their geocoded equivalent coordinates could be passed to the `distance_matrix` method. The following example first initializes Q's house address and the treatment center address, after which the `distance_matrix` method calculates (or retrieves) data about the route:

```

# manually initialize subject home address coords
pt_lat = 32.6848432

```

```

pt_lon = -117.1300151
coords_origin = (pt_lat, pt_lon)

# manually initialize National City treatment center address
center_lat = 32.6763009
center_lon = -117.085868
coords_dest = (center_lat, center_lon)

# initialize the Google Maps key
gmaps_key = googlemaps.Client(key="YOUR_SECRET_KEY_GOES_HERE")

# calculate driving distance
dist = gmaps_key.distance_matrix(coords_origin,
                                  coords_dest,
                                  mode='driving')

```

The data structure of the resultant **dist** dictionary is demonstrated:

```

print(dist)

{'destination_addresses': ['1900 E Plaza Blvd, National City, CA 91950, USA'],
 'origin_addresses': ['3455 Senn St, San Diego, CA 92136, USA'], 'rows': [{'elements':
[{'distance': {'text': '5.1 km', 'value': 5142}, 'duration': {'text': '11 mins',
'value': 675}, 'status': 'OK'}]}], 'status': 'OK'}

```

Note that the most precise distance is returned from the method in meters in the **value** key, so the following code extracts only **value**:

```

print(dist['rows'][0]['elements'][0]['distance']['value'])

5142

```

Equivalent results can be calculated instead by passing street addresses, rather than latitude/longitude coordinates, although this method is not further explored in this text:

```

# this example demonstrates the equivalent calculation instead using street addresses
pt_addy = 'Naval Base San Diego Bldg #3526, San Diego, CA 92136'

center1_addy = '1900 E Plaza Blvd, National City, CA 91950'

dist_v2 = gmaps_key.distance_matrix(pt_addy,
                                     center1_addy,
                                     mode='driving')

print(dist_v2['rows'][0]['elements'][0]['distance']['value'])

5126

```

With this quick introduction to **distance_matrix**, a Python function can be engineered to return the driving distance (in miles, rounded to the nearest tenth of a mile) between two locations:

```

# calculate driving distance between two locations
def get_distance(lat_start, lon_start, lat_end, lon_end):
    coords_origin = (lat_start, lon_start)
    coords_dest = (lat_end, lon_end)
    # generates distances in meters
    dist = gmaps_key.distance_matrix(coords_origin, coords_dest,
                                     mode='driving')['rows'][0]['elements'][0]['distance']['value']
    # converts meters to miles
    dist = round(dist * 0.000621371, 1)
    return dist

```

Thus, when the **get_distance** Python function is called and passed the coordinates for Q's lodging and the National City In-N-Out, the distance is again shown to be 3.2 miles:

```

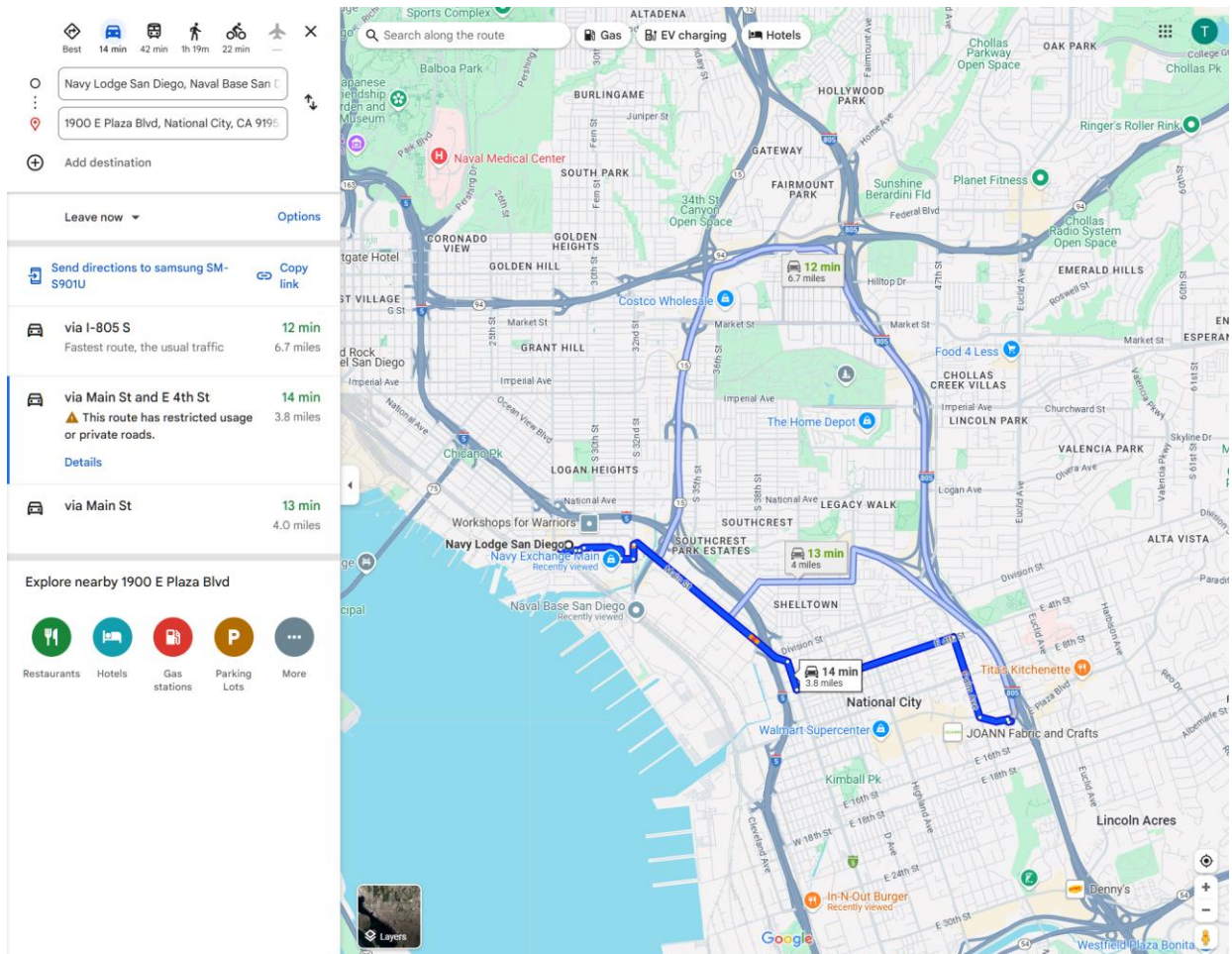
q_dist = get_distance(pt_lat, pt_lon, center_lat, center_lon)

```

```
print(q_dist)
```

3.2

This is nearly identical to the route distance produced through a manual Google Maps query.



The `get_distance` user-defined Python function is now operational, but must be modified so it can be called in SAS through the FCMP procedure. The following program file should be saved as `get_distance.py`, noting that the `Output` statement was again required so that the `dist_key` key would be passed back to the SAS session:

```
# import the Google Maps package
import googlemaps

# initialize the Google Maps key
gmaps_key = googlemaps.Client(key="YOUR_SECRET_KEY_GOES_HERE")

# calculate driving distance between two locations
def get_distance(lat_start, lon_start, lat_end, lon_end):
    "Output: dist_key"
    coords_origin = (lat_start, lon_start)
    coords_dest = (lat_end, lon_end)
    # generates distances in meters
    dist = gmaps_key.distance_matrix(coords_origin, coords_dest,
                                     mode='driving')['rows'][0]['elements'][0]['distance']['value']
    # converts meters to miles
    dist = round(dist * 0.000621371, 1)
    return dist
```

And with the `get_distance.py` program file saved, the user-defined SAS function `sas_get_dist` can call the `get_distance` Python function:

```
proc fcmp outlib = work.funcs.py;
  function sas_get_dist(origin_lat, origin_lon, dest_lat, dest_lon);
    declare object py2(python);
    rc = py2.infile("&loc_file.get_distance.py");
    rc = py2.publish();
    rc = py2.call("get_distance", origin_lat, origin_lon, dest_lat, dest_lon);
    dist_miles = py2.results["dist_key"];
    return(dist_miles);
  endfunc;
quit;
```

Because only one value (distance) is returned, a user-defined function is declared using the `FUNCTION` statement, as opposed to the previous declaration of a subroutine using the `SUBROUTINE` statement. Similarly, in lieu of relying on the `OUTARGS` statement to pass values *by reference* (to the calling program, the `DATA` step), the `RETURN` statement now returns `dist_miles` to the calling program *by value*.

Note that Python Component Object names must be unique within a SAS session, so whereas `py` was declared in the `sas_get_coords` subroutine, `py2` is declared in the `sas_get_dist` function.

The following `DATA` step demonstrates calling the `sas_get_coords` subroutine to geocode Q's address, calling the `sas_get_coords` subroutine to geocode the National City In-N-Out's address, and calling the `sas_get_dist` function to calculate the distance between these two locations:

```
data test_distance;
  length pt_addy center_addy $100 pt_lat pt_lon center_lat center_lon dist_miles 8;
  pt_addy = 'Naval Base San Diego Bldg #3526, San Diego, CA 92136';
  center_addy = '1900 E Plaza Blvd, National City, CA 91950';
  call missing(pt_lat, pt_lon, center_lat, center_lon, dist_miles);

  * geocode subject address;
  call sas_get_coords(pt_addy, pt_lat, pt_lon);

  * geocode center address;
  call sas_get_coords(center_addy, center_lat, center_lon);

  * calculate distance between origin and destination coords;
  dist_miles = sas_get_dist(pt_lat, pt_lon, center_lat, center_lon);
  put pt_lat= pt_lon= center_lat= center_lon= dist_miles=;
run;

pt_lat=32.6848432 pt_lon=-117.1300151 center_lat=32.6763009 center_lon=-117.085868
dist_miles=3.2
```

The results are identical to calling the Python functions directly in Python, and demonstrate that Q needs to drive 3.2 miles to reach his primary treatment center. In a more complete demonstration of this scenario, subject treatment data would likely be maintained across multiple observations, with each observation mapped to the associated treatment center for that visit, facilitating the longitudinal analysis of distance-to-treatment for each subject.

Moreover, this `DATA` step demonstrated that all subject and center data can be maintained in SAS, and yet passed to (and transformed by) user-defined Python functions that are called by `FCMP` user-defined functions and subroutines. Thus, SAS practitioners are able to leverage existent user-defined Python functions and, by modifying only one line of code in each function, to call them from the `DATA` step.

CONCLUSION

By leveraging the power and flexibility of the `FCMP` procedure, SAS practitioners do not need to abandon the comfort of developing in the Base SAS language, and can augment their SAS experience by calling Python functions using the native Python interpreter of their choice. For CROs and other organizations who may rely almost exclusively on SAS software, and yet who are concerned that they are missing out on

bleeding-edge, open-source advancements and solutions (such as the Google Maps Platform and its expansive API library), the FCMP procedure effortlessly bridges this gap—by delivering and maximizing software interoperability and data transferability.

REFERENCES

- GitHub. (2025). *Google Maps GitHub*. Retrieved from google-maps-services-python: <https://github.com/googlemaps/google-maps-services-python/tree/master/googlemaps>
- GitHub. (2025). *Google Maps GitHub Repository*. Retrieved from Geocoding.py: <https://github.com/googlemaps/google-maps-services-python/blob/master/googlemaps/geocoding.py>
- GitHub. (2025). *Google Maps GitHub Repository*. Retrieved from distance_matrix.py: https://github.com/googlemaps/google-maps-services-python/blob/master/googlemaps/distance_matrix.py
- Google. (2025, 4 1). *Client Libraries for Google Maps Web Services*. Retrieved from Google Maps Platform: <https://developers.google.com/maps/web-services/client-library>
- Google. (2025). *Google Maps Platform*. Retrieved from Google Maps Platform: <https://developers.google.com/maps>
- Google. (2025). *Google Maps Platform*. Retrieved from APIs and Services: <https://console.cloud.google.com/google/maps-apis/api-list>
- Google. (2025). *Google Maps Platform*. Retrieved from Geocoding API: <https://developers.google.com/maps/documentation/geocoding>
- Google. (2025). *Google Maps Platform*. Retrieved from Client Libraries for Google Maps Web Services: <https://developers.google.com/maps/documentation/geocoding/client-library>
- googlemaps / google-maps-services-python*. (2025, 4 1). Retrieved from GitHub: <https://github.com/googlemaps/google-maps-services-python>
- google-maps-services-python/googlemaps*. (2025, 4 1). Retrieved from GitHub: <https://github.com/googlemaps/google-maps-services-python/blob/master/googlemaps/geocoding.py>
- Hughes, T. M. (2016). *SAS® Data Analytic Development: Dimensions of Software Quality*. Hoboken, NJ: John Wiley and Sons.
- Hughes, T. M. (2023). Make You Holla' Tikka Masala: Creating User-Defined InformatS Using the PROC FORMAT OTHER Option To Call User-Defined FCMP Functions That Facilitate Data Ingestion Data Quality. *PharmaSUG*. San Francisco, CA. Retrieved from <https://www.lexjansen.com/pharmasug/2023/AP/PharmaSUG-2023-AP-291.pdf>
- Hughes, T. M. (2023). *SAS® Data-Driven Development: From Abstract Design to Dynamic Functionality, Second Edition*. San Diego: Kindle Direct Publishing.
- Hughes, T. M. (2023). Sorting a Bajillion Variables—When SORTC and SORTN Subroutines Have Stopped Satisfying, User-Defined PROC FCMP Subroutines Can Leverage the Hash Object to Reorder Limitless Arrays. *PharmaSUG*. San Francisco, CA. Retrieved from <https://www.lexjansen.com/pharmasug/2023/AP/PharmaSUG-2023-AP-094.pdf>
- Hughes, T. M. (2023). Undo SAS® Fetters with Getters and Setters: Supplanting Macro Variables with More Flexible, Robust PROC FCMP User-Defined Functions That Perform In-Memory Lookup and Initialization Operations. *PharmaSUG*. San Francisco, CA. Retrieved from <https://www.lexjansen.com/pharmasug/2023/HT/PharmaSUG-2023-HT-093.pdf>
- Hughes, T. M. (2026). *PROC FCMP User-Defined Functions: An Introduction to the SAS® Function Compiler, Second Edition*.
- NASA. (2024, March 24). *Uranus Facts*. Retrieved from <https://science.nasa.gov/uranus/facts/>

SAS Institute. (2025). *SAS® 9.4 Administration*. Retrieved from Configuring SAS to Run the Python Language:
<https://go.documentation.sas.com/doc/en/bicdc/9.4/biasag/n1mqxnmfmfu83en1if8icqmx8cdf.htm>

SAS Institute. (2025). *SAS® 9.4 and SAS® Viya® 3.5 Programming Documentation*. Retrieved from Using PROC FCMP Python Objects:
https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/lecompobjref/p18qp136f91aaqn1h54v3b6pkant.htm

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Name: Troy Martin Hughes

E-mail: troymartinhughes@gmail.com