

**Building a Better Mousetrap:
Automating Personalized Custom Commute Planning
via Python & Google APIs**

Sharon W.H. Ling

sling35@gatech.edu

Dual Masters Student in
City Planning & Civil Engineering (Transportation)

CP 6581: Programming for GIS
Final Project Report

Georgia Institute of Technology
November 20, 2019

Introduction

I currently work part-time as a Transportation Coordinator attached to the in-house Commutes Team of Company A, which is headquartered in Downtown Atlanta, GA. The Commutes Team is tasked with encouraging Company A's employees to take alternative transportation to work, i.e. walking / biking / public transit / other options that reduce or avoid single-occupancy vehicle (SOV) use.

To this end, 'personalized custom commute planning' is one of the key services that the Commutes Team offers to Company A's employees. For example, an employee who is considering switching from driving to alternative transportation can ask the Commutes Team for advice on the 'best option' for taking transit to / from work (while providing their home address and desired time of arrival at / departure from Company A on a regular workday).

In response, the Commutes Team is expected to reply the said employee within 3 business days with a personalized 'custom commute plan'. The latter is essentially an information sheet that details (1) the recommended overall route using transit, (2) a breakdown of commute instructions / directions for each individual segment of the trip, (3) route maps for the individual segments of the trip, (4) a schedule of expected departure / arrival times for the individual segments of the trip, and (5) information on payment options for the entire trip.

The idea behind the 'personalized custom commute planning' service is that Company A employees who are unfamiliar with transit options would benefit from the expertise of the Commutes Team, whose members would presumably be more familiar with transit in Atlanta and can advise accordingly on what is the 'best option' to take. This is certainly a noble goal; however, the processes behind this service can stand to be improved.

Currently, the Commute Team's SOP is to look up transit options *manually* on Google Maps and type up relevant information / navigational instructions (i.e.: "drive from [employee's home address] to North Springs MARTA, take the Red Line to Civic Center MARTA Station, board Company A's shuttle from Civic Center MARTA Station to Company A's headquarters; approximate departure and arrival times for the Red Line are [etc]"). The Commutes Team then creates the 'custom commute plan' from the compiled transit information as per above.

Unfortunately, the entire process is fairly slow and tedious, especially when Commutes Team staff are forced to repeatedly evaluate and write up information from different Google Maps / browser search results over to the 'custom commute plan' template. The process is

also susceptible to human error – more so when Commutes Team staff have to manually type in information, or manually calculate odd time values to populate the ‘custom commute plan’ with (e.g ‘If employee X arrives at Civic Center MARTA at 8:38am and immediately boards Company A’s shuttle which takes 15 minutes to arrive at Company A, what is the final arrival time?’). These issues slow down, complicate and frustrate the processes involved; as a result, the creation of a ‘custom commute plan’ can take anywhere from 1.5 to 2.5 hours at current capacity. This hampers the Commutes Team’s ability to handle more than a handful of requests within a short timeframe, especially when these requests are competing with other, more urgent tasks that demand the Commutes Team’s attention.

In addition, the custom commute plans could stand to be enhanced in terms of the information and insights contained, so that employees are able to make better decisions about their commuting choices and perhaps even resolve any lingering doubts about taking transit. However, the Commutes Team is currently unable to deliver this partly due to the amount of time exhausted on manually Googling and compiling results alone, which leaves little time for anything else. Some insights that would be particularly valuable, but are currently not provided, include and are not limited to (1) the *optimum* time for departures / arrivals within a particular timeframe (given that traffic conditions vary greatly depending on the time of the day, (2) the *comparative* expected travel times for driving versus transit given a particular departure / arrival time, and (3) the *total* door-to-door travel time for overall trips as well as individual segments of said trips. These insights, if analyzed and made available, would surely help the Commutes Team and employees plan more efficient and comfortable journeys via transit.

In the long run, my aim is to automate and enhance Company A’s personalized custom commute planning service as much as possible. I foresee a boost in time savings, productivity, and satisfaction for both the Commutes Team and employees who utilize this service. Towards this goal, I have taken it upon myself to build a Custom Commute Planner (version 1.0 as of November 20, 2019) using Python 2.7 and with the Pyscripter IDE, with details as follows.

Data Sources

Google Maps Platform is a set of APIs that allows developers to retrieve data from Google Maps. This includes the Directions API, which is a service that calculates directions between locations and can search for directions for several modes of transportation, including transit, driving, walking, or cycling. According to the [Google Directions API FAQ](#), the process works by having “[the user] access the Directions API through an HTTP interface, with requests constructed as a URL string, using text strings or latitude/longitude coordinates to

identify the locations, along with [the user's] API key.” On the subject of the API key, the FAQ further notes that “to use the Directions API, [the user] must first activate the API in the Google Cloud Platform Console and obtain the proper authentication credentials. [The user needs] to provide an API key in each request.”

Once a search query is submitted with the relevant parameters (at the bare minimum, the user must provide the start address, end address and travel mode for the intended trip), the Directions API returns a response in JSON format. The response typically contains, but is not limited to, the following information:

- Start address for the overall trip – previously provided by the user
- End address for the overall trip – previously provided by the user
- Travel mode for the overall trip – previously provided by the user
- Departure time for the overall trip – may or may not be provided by the user
- Arrival time for the overall trip – may or may not be provided by the user
- Distance for the overall trip
- Duration for the overall trip
- Individual ‘steps’ for the trip i.e. trip segments, and corresponding information:
 - Travel instructions for the overall segment (e.g. ‘Walk to XX’)
 - Start address / location for the segment*
 - End address / location for the segment*
 - Distance for the segment
 - Duration for the segment
 - Travel mode for the segment
 - Individual sub-steps for the segment, and corresponding information down to the last level of detail (e.g. travel instructions ‘Turn left at...’, start location, end location, distance, duration, travel mode for the sub-step); these results are usually associated with a travel mode other than transit e.g. ‘walking’

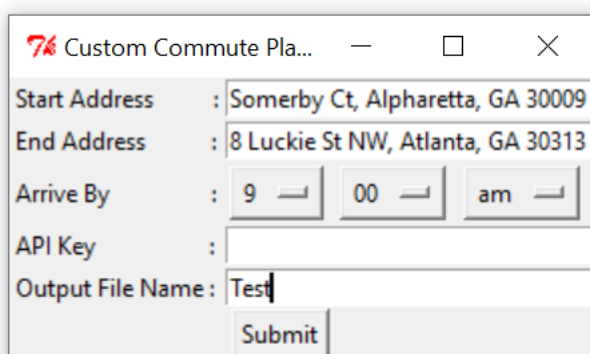
** Results vary according to whether the segment’s travel mode is identified as ‘transit’ or a mode other than transit e.g. ‘walking’; interestingly, ‘transit’ segments will return a ‘start address’ and ‘end address’ information in the form of a text string for each, while ‘non-transit’ segments will return ‘start location’ and ‘end location’ information in the form of latitude and longitude values without any corresponding addresses.*

To test the Custom Commute Planner, I opted not to use any existing employee's home address due to privacy concerns, but instead randomly picked a residential location in Atlanta (i.e. '1100 Somerby Ct, Alpharetta, GA 30009') as a start address. Meanwhile, the approximate location of Company A is used for the end address. Using this particular start address and end address, I performed a [preliminary manual search with Google Maps](#) and noted that the queried transit trip results in a suggested trip of multiple segments and modes inclusive of transit (bus/rail) and walking (see Appendix A). Based on these results, I tested and fine-tuned the Custom Commute Planner to make sure that it would return similar results when performing the same query as coded in Python 2.7.

Methodology and Results

The full working code for the Custom Commute Planner is contained in Appendix B. When running the code, users (i.e. the Commutes Team for now) will be prompted to input the necessary information into a custom user interface created via Tkinter (see Figure 1 below). For now, the information required comprises: start address (i.e. the employee's home address), end address (i.e. Company A's office address), intended time of arrival, user's API key, and the intended name of the text file that the Custom Commute Planner will save the formatted results of the search query to. The overall travel mode is pre-defined as 'transit' within the code, so there is no need for the user to specify this information.

Figure 1: Custom user interface for the Custom Commute Planner (API key left blank)



Start Address	: Somerby Ct, Alpharetta, GA 30009
End Address	: 8 Luckie St NW, Atlanta, GA 30313
Arrive By	: 9 :00 am
API Key	:
Output File Name	: Test

Submit

When the user presses 'Submit', the code submits a search query to Directions API, and returns the corresponding JSON results (see Appendix C). The code then formats the JSON results into a format that is more readable and useful for users (see Appendix D). In addition, the results are printed to both the Pyscripter console and also a text file on the user's computer desktop (intended for easy retrieval by the Commutes Team).

Challenges

Overall, developing the full working code was not without its challenges – especially when attempting to interpret and format the JSON results into a desired format that would be helpful for the Commutes Team and employees. For example, it seemed crucial to provide not just a quick summary of the overall commute, but to provide details for each individual segment so that the employee in question might have a sense of how long and how far they would need to walk or take the bus/train, and at what time they would have to depart or arrive. However, attempts to extract and directly display these details from the initial JSON results were initially unsuccessful for some segments but strangely successful for others. Upon closer examination, I found that the reason for this was due to inconsistent output generated in the JSON results.

For example, the JSON results for the *overall* trip typically contain fundamental details such as the departure and arrival times, but these details are *not* generated for individual segments of the trip (referred to as ‘steps’ in the JSON results) that are undertaken via a mode other than transit (e.g. walking). This is likely because Google Maps pulls the departure and arrival time information for transit trips (bus/rail) from existing GTFS schedules, while there is no fixed schedule per se to pull this information from for non-bus/rail trips. However, because ‘duration’ values are calculated and returned in JSON results for all individual trip segments, it is possible to derive corresponding departure and arrival times for non-transit segments pre- or post-transit segments by performing addition or subtraction using Python code (and then converting those values, which are in UTC, to the corresponding time in user’s local time zone). In a similar vein, JSON results for ‘transit’ segments return a ‘start address’ and ‘end address’ information in the form of a text string for each, while ‘non-transit’ segments will return ‘start location’ and ‘end location’ information in the form of latitude and longitude values without any corresponding addresses. Therefore, to derive a ‘proper’ address for either the said start or end location of a non-transit trip segment, geocoding queries must be made using the latitude and longitude values returned by the initial JSON results.

Since the intention was to ensure consistency of formatting for the output printed to the Pyscripter console and to text file, there was no choice but to expend effort on designing workarounds for these issues within the Custom Commute Planner’s code (as noted above). On one hand, this allowed me the opportunity to become very familiar with the JSON results and figure out ways to format these results via code for the Commutes Team’s benefit. On the other hand, the development of these workarounds consumed much more time than intended.

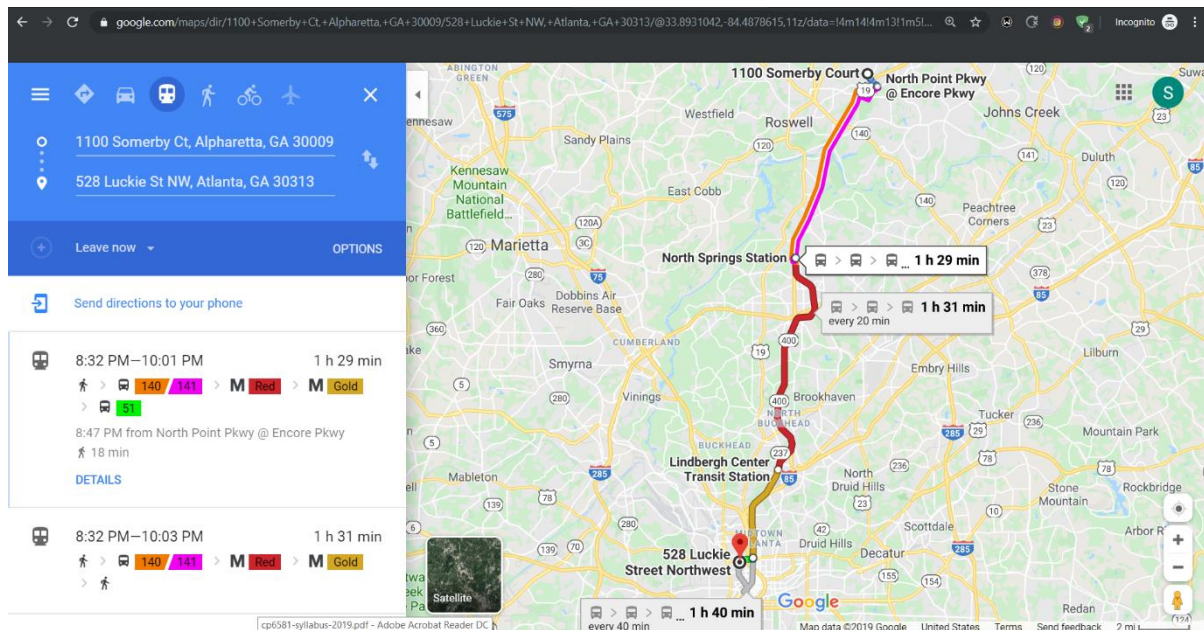
Potential for Improvement

Due to time limitations and the time spent on developing workarounds for the issues encountered as described above, it was not possible to work on developing more sophisticated features for the Custom Commute Planner as originally intended. However, I plan to continue working on this project even after the submission of this report. Some of the features that I would like to code and add into the Custom Commute Planner include:

- The ability to piece together a comprehensive trip that is composed of driving **and** transit modes, not solely a trip by transit (e.g. in the event that an employee lives too far away to walk to a bus stop or train station, and would have to drive there first instead). There are many such cases among employees at Company A. Currently, Google Maps does not offer the functionality to incorporate driving within its transit mode search option; it simply returns an error message that says ‘*Sorry, we could not calculate transit directions from [Start Address] to [End Address]*’ if it there are no nearby transit options. This would likely require the code to piece together search results from multiple queries (e.g. search for Park-and-Ride sites within a certain radius of the start address, return trip details, then search for transit options from the Park-and-Ride sites to Company A, etc).
- A ‘Depart By’ set of buttons in the custom user interface, for users to key in intended Departure Time rather than Arrival Time if they so choose (at present, the Custom Commute Planner is set up to receive arrival time input only);
- An optional ‘Day of the Week’ dropdown menu in the custom user interface, for users to key in the intended day of the week that they plan to take their trip. This feature might be useful if we assume that commute patterns are different on certain weekdays e.g. Tuesday versus Friday, or if there are users who would like to come in to work on Saturdays. This would likely require the code (which defaults to the current date/day) to select the chosen day within the current week;
- A ‘Driving vs Transit’ feature that would return trip information on a driving versus transit trip, especially in terms of expected departure time, arrival time, and duration. This might be useful for users who would like to know how much time they might save (or need to budget extra for, unfortunately) if taking transit instead of driving;

- Expanded results in the form of a timetable, and even a graph, of 'best and worst times of the day to commute' based on the user's submitted information. This would likely require multiple and successive search queries to be performed by code, with multiple departure times incremented by 30 minutes with each search query (e.g. 6:00am, 6:30am, 7:00am, etc); and
- Last but not least, an in-house app made available to Company A employees who would like to make their own search queries and view the results themselves, rather than having the Commutes Team be the middleperson each time.

Appendix A: Results of a manual GoogleMaps search with user's information



Appendix B: Full working code for the Custom Commute Planner

Step 1: import all necessary Python packages

```
import googlemaps, pendulum, tinyurl # install via pip method if unable to import directly through
Pyscripter
```

```
import Tkinter as Tk
```

```
from Tkinter import *
```

```
import os, sys, json, time, urllib
```

```
from urllib import urlencode
```

```
import collections
```

```
from collections import OrderedDict # this keeps the order of the 'dictionary' especially when
submitting URLs
```

Step 2: setup a custom user interface created with Tkinter

```
# User inputs desired Start Address, End Address, Arrive By [hour (HH), minute (MM), time of day
(AM/PM)], API Key, and Output File Name into custom user interface
```

```
# User input values are collected and carried over to Step 3
```

```
root = Tk()
```

```
root.title("Custom Commute Planner v 1.0")
```

```
# set "Start Address" entry in custom user interface
```

```

Label(root, text = "Start Address      :").grid(row = 0, sticky = W)
startadd = Entry(root, width = 30)
startadd.grid(row = 0, column = 1, columnspan = 3, sticky = W)

# set "End Address" entry in custom user interface
Label(root, text = "End Address      :").grid(row = 1, sticky = W)
endadd = Entry(root, width = 30)
endadd.grid(row = 1, column = 1, columnspan = 3, sticky = W)

# set "Arrive By" entries in custom user interface
Label(root, text = "Arrive By      :").grid(row = 2, sticky = W)

# set "Arrive By" hour (HH) value in custom user interface
hh_options = [1,2,3,4,5,6,7,8,9,10,11,12]
hh = StringVar(root)
hh.set(hh_options[0]) # default value

a = OptionMenu(root, hh, *hh_options)
a.grid(row = 2, column = 1, sticky = W)

# set "Arrive By" minute (MM) value in custom user interface
mm_options = ["00", "15", "30", "45"]
mm = StringVar(root)
mm.set(mm_options[0]) # default value

b = OptionMenu(root, mm, *mm_options)
b.grid(row = 2, column = 2, sticky = W)

# set "Arrive by" time of day (AM/PM) value in custom user interface
ampm_options = ["am", "pm"]
ampm = StringVar(root)
ampm.set(ampm_options[0]) # default value

c = OptionMenu(root, ampm, *ampm_options)
c.grid(row = 2, column = 3, sticky = W)

# set "API Key" entry in custom user interface
Label(root, text = "API Key      :").grid(row = 4, sticky = W)
apikey = Entry(root, width = 30)
apikey.grid(row = 4, column = 1, columnspan = 3, sticky = W)

```

```

# set "Output File Name" entry in custom user interface
Label(root, text = "Output File Name :").grid(row = 5, sticky = W)
OutputName = Entry(root, width = 30)
OutputName.grid(row = 5, column = 1, columnspan = 3, sticky = W)

# Step 3: gather user input values for further action

def getInput():

    start = startadd.get()
    end = endadd.get()
    HH = hh.get()
    MM = mm.get()
    AMPM = ampm.get()
    API_KEY = apikey.get()
    OUTPUT_NAME = OutputName.get()
    root.destroy() #closes custom user interface after Submit button is clicked

    global UserInput
    UserInput = [start, end, HH, MM, AMPM, API_KEY, OUTPUT_NAME] #collects user input into a list

Button(root, text="Submit", command=getInput).grid(row = 6, column = 1)

mainloop()

# assign labels for user input values
start = UserInput[0]
end = UserInput[1]
HH = UserInput[2]
MM = UserInput[3]
AMPM = UserInput[4]
API_KEY = UserInput[5]
OUTPUT_NAME = UserInput[6]

# create a class that will print output to both the console and a file (on the user's desktop)
# source: https://stackoverflow.com/questions/24204898/python-output-on-both-console-and-file/24206109#24206109

desktop = os.path.expanduser("~/Desktop") # set user's desktop as output path

```

```

class Printer:
    def __init__(self, filename):
        self.out_file = open(filename, "w")
        self.old_stdout = sys.stdout
        #this object will take over stdout's job
        sys.stdout = self
    #executed when the user does a 'print'
    def write(self, text):
        self.old_stdout.write(text)
        self.out_file.write(text)
    #executed when 'with' block begins
    def __enter__(self):
        return self
    #executed when 'with' block ends
    def __exit__(self, type, value, traceback):
        #we don't want to log anymore. Restore the original stdout object.
        sys.stdout = self.old_stdout

# Step 4: print user input to both the console and a file (on the user's desktop)

with Printer(desktop+"\\\\"+OUTPUT_NAME+".txt"):
    print "USER INPUT    :"
    print
    print "Start Address  : " + start
    print "End Address    : " + end
    print "Arrive By      : " + str(HH) + ":" + str(MM) + AMPM
    print

# Step 5: send user input values to json/Google Maps

if HH == "12" and AMPM == "am": # effectively convert HH entries that are 12:00am to 00:00 am
    HH = 0

if HH != "12" and AMPM == "pm": # effectively convert HH entries that are PM to 24-hour time i.e.
1:00pm = 13:00
    HH = int(HH)+12
else:
    HH = int(HH)

```

```

# convert HH, MM and AM/PM user input to UTC values for json / GoogleMaps submission
# pendulum.today() returns today's date with HH & MM automatically set at 00:00; so user input of HH
& MM time is added to that
    dt = pendulum.today().add(hours=int(HH)).add(minutes=int(MM))
    dt = int(dt.timestamp())

# generate json submission url for general inquiry
    jsonurl_prefix = 'https://maps.googleapis.com/maps/api/directions/json?'
    jsonurl_data = urllib.urlencode(OrderedDict([("origin", start), ("destination", end), ("arrival_time",
str(dt)), ("mode","transit"), ("key", API_KEY)]))
    jsonurl = jsonurl_prefix+jsonurl_data
    jsonurl_gresp = urllib.urlopen(jsonurl)
    jsonurl_jresp = json.loads(jsonurl_gresp.read())
    jsonurl_url = tinyurl.create_one(jsonurl)

# generate a unique GoogleMaps map link for that individual segment
# unfortunately, the link defaults to "depart now" and departure/arrival times can't be set -- this seems
to be a Google Maps quirk
    gmapsurl_prefix = 'https://www.google.com/maps/dir/?api=1&'
    gmapsurl_data = urllib.urlencode(OrderedDict([("origin", start), ("destination", end), ("travelmode",
"transit")]))
    gmapsurl = gmapsurl_prefix+gmapsurl_data
    gmapsurl_url = tinyurl.create_one(gmapsurl)

# generate latlong submission url for latlong inquiry --- the latlong details are called upon further down
the code
    latlngurl_prefix = 'https://maps.googleapis.com/maps/api/geocode/json?latlng='

# Step 6: print output to both the console and a file (on the user's desktop)

    print "SUMMARY OUTPUT  :"
    print
    print "Google Maps URL  :", gmapsurl_url
    print "Total Distance  :", jsonurl_jresp['routes'][0]['legs'][0]['distance']['text']
    print "Total Duration  :", jsonurl_jresp['routes'][0]['legs'][0]['duration']['text']
    print "Departure Time  :", jsonurl_jresp['routes'][0]['legs'][0]['departure_time']['text']
    print "Arrival Time   :", jsonurl_jresp['routes'][0]['legs'][0]['arrival_time']['text']
    print
    print "DIRECTIONS      :"
```

```

for i in range (0, len (jsonurl_jresp['routes'][0]['legs'][0]['steps'])):
    time.sleep(.1)
    i_ins = jsonurl_jresp['routes'][0]['legs'][0]['steps'][i]['html_instructions'] # sometimes can be
gibberish so supplement with proper From/To details
    i_mod = jsonurl_jresp['routes'][0]['legs'][0]['steps'][i]['travel_mode']
    i_dis = jsonurl_jresp['routes'][0]['legs'][0]['steps'][i]['distance']['text']
    i_dur = jsonurl_jresp['routes'][0]['legs'][0]['steps'][i]['duration']['text']
    print
    print('Trip Segment {0}'.format(i+1)), " : ", i_ins
    print "Travel Mode      : " + i_mod.title()

# print From/To details for individual segments of the trip
# note: this is tricky because of how Google Maps' json output is generated even when submitting an
overall transit-based query:
    # 1) if a trip segment is identified as "transit", the json output includes a ['departure stop'] and
['arrival stop'] address string
    # 2) but if a trip segment is identified as non-transit i.e. walking, the json output only provides
lat/longs of the start/end location
    # 3) so for a non-transit trip segment, to produce a departure/arrival stop address as json output, a
reverse geocoding query must be submitted
    # since the first segment of the overall trip is likely non-transit i.e. walk from home to bus stop,
    # check that "transit_details" are NOT in the json output and then proceed to the reverse
geocoding query to create From/To details
    if "transit_details" not in jsonurl_jresp['routes'][0]['legs'][0]['steps'][i]:
        i_start_lat = jsonurl_jresp['routes'][0]['legs'][0]['steps'][i]['start_location']['lat']
        i_start_lng = jsonurl_jresp['routes'][0]['legs'][0]['steps'][i]['start_location']['lng']
        i_start_latlngurl = latlngurl_prefix+str(i_start_lat)+","+str(i_start_lng)+"&key="+API_KEY
        i_start_latlngurl_gresp = urllib.urlopen(i_start_latlngurl)
        i_start_latlngurl_jresp = json.loads(i_start_latlngurl_gresp.read())
        i_start = i_start_latlngurl_jresp['results'][0]['formatted_address']
        print "From          : ", i_start
        i_end_lat  = jsonurl_jresp['routes'][0]['legs'][0]['steps'][i]['end_location']['lat']
        i_end_lng  = jsonurl_jresp['routes'][0]['legs'][0]['steps'][i]['end_location']['lng']
        i_end_latlngurl = latlngurl_prefix+str(i_end_lat)+","+str(i_end_lng)+"&key="+API_KEY
        i_end_latlngurl_gresp = urllib.urlopen(i_end_latlngurl)
        i_end_latlngurl_jresp = json.loads(i_end_latlngurl_gresp.read())
        i_end = i_end_latlngurl_jresp['results'][0]['formatted_address']
        print "To          : ", i_end
    # generate a unique GoogleMaps map link for that individual segment

```

unfortunately, the link's departure/arrival times can't be set -- this seems to be a Google Maps quirk

```
gmapsurl_data = urllib.urlencode(OrderedDict([("origin", i_start), ("destination", i_end),
("travelmode", i_mod.lower())]))
gmapsurl = gmapsurl_prefix+gmapsurl_data
gmapsurl_url = tinyurl.create_one(gmapsurl)
print "Google Maps URL : " + gmapsurl_url

# alternatively, if the segment IS identified as transit, retrieve ['departure stop'] and ['arrival stop']
address strings for From/To details:
```

```
if "transit_details" in jsonurl_jresp['routes'][0]['legs'][0]['steps'][i]:
    i_start = jsonurl_jresp['routes'][0]['legs'][0]['steps'][i]['transit_details']['departure_stop']['name']
    print "From      : " + i_start
    i_end = jsonurl_jresp['routes'][0]['legs'][0]['steps'][i]['transit_details']['arrival_stop']['name']
    print "To        : " + i_end
```

generate a unique GoogleMaps map link for that individual segment

unfortunately, the link defaults to "depart now" and departure/arrival times can't be set -- this seems to be a Google Maps quirk

```
gmapsurl_data = urllib.urlencode(OrderedDict([("origin", i_start), ("destination", i_end),
("travelmode", i_mod.lower())]))
gmapsurl = gmapsurl_prefix+gmapsurl_data
gmapsurl_url = tinyurl.create_one(gmapsurl)
print "Google Maps URL : " + gmapsurl_url
print "Distance      : " + i_dis
print "Duration      : " + i_dur
```

print Departure Time & Arrival Time details for individual segments of the trip

note: this is tricky because of how Google Maps' json output is generated even when submitting an overall transit-based query:

1) if a trip segment is identified as "transit", the json output includes a ['departure time'] and ['arrival time'] string

2) but if a trip segment is identified as non-transit i.e. walking, the json output does not contain this information

3) so for a non-transit trip segment, to produce departure time & arrival time output, these values must often be calculated manually

4) we can also assume that for individual non-transit trip segments,

a) where i = 0 (aka 1st trip segment), departure time = departure time of overall trip & arrival time = departure time of next trip segment

b) where i > 0 (aka 2nd or more trip segment), departure time = arrival time of previous trip segment & arrival time = departure time of next trip segment

c) arrival time = departure time + duration of trip segment (which IS produced as json output)

```

# this leads to the following code:
# where i = 0 (aka 1st trip segment)
if i == 0:
    print "Departure Time  : " + jsonurl_jresp['routes'][0]['legs'][0]['departure_time']['text']
    i_arr = (jsonurl_jresp['routes'][0]['legs'][0]['departure_time']['value']) +
(jsonurl_jresp['routes'][0]['legs'][0]['steps'][i]['duration']['value'])
    i_arr = pendulum.from_timestamp(i_arr, tz='local')
    i_arr = i_arr.to_day_datetime_string()
    i_arr = str(i_arr).split()
    print "Arrival Time   : " + i_arr[4] + i_arr[5].lower()
# where i > 0 (aka 2nd or more trip segment)
if i is not 0 and jsonurl_jresp['routes'][0]['legs'][0]['steps'][i]['travel_mode'] == "WALKING":
    print "Departure Time  : " + jsonurl_jresp['routes'][0]['legs'][0]['steps'][i-
1]['transit_details']['arrival_time']['text']
    i_arr = (jsonurl_jresp['routes'][0]['legs'][0]['steps'][i-1]['transit_details']['arrival_time']['value']) +
(jsonurl_jresp['routes'][0]['legs'][0]['steps'][i]['duration']['value'])
    i_arr = pendulum.from_timestamp(i_arr, tz='local')
    i_arr = i_arr.to_day_datetime_string()
    i_arr = str(i_arr).split()
    print "Arrival Time   : " + i_arr[4] + i_arr[5].lower()
# where the individual trip segment is identified as "transit"
if "transit_details" in jsonurl_jresp['routes'][0]['legs'][0]['steps'][i]:
    i_dep = jsonurl_jresp['routes'][0]['legs'][0]['steps'][i]['transit_details']['departure_time']['text']
    print "Departure Time  : " + i_dep
    i_arr = jsonurl_jresp['routes'][0]['legs'][0]['steps'][i]['transit_details']['arrival_time']['text']
    print "Arrival Time   : " + i_arr
print
print "Thank you for using this Custom Commute Planner v 1.0!"

```

Appendix C: JSON URL produced by the submitted search query

https://maps.googleapis.com/maps/api/directions/json?origin=1100+Somerby+Ct%2C+Alpha+retta%2C+GA+30009&destination=528+Luckie+St+NW%2C+Atlanta%2C+GA+30313&arrival_time=1574258400&mode=transit&key=AlzaSyCb_menILhJWN3KluLP8U2EJKtgyVJPIsY, also accessible via <http://tinyurl.com/rwzh6a8> (converted to tinyurl format by code)

Appendix D: Formatted JSON results printed to Pyscripter console and text file

USER INPUT :

Start Address : 1100 Somerby Ct, Alpharetta, GA 30009
End Address : 528 Luckie St NW, Atlanta, GA 30313
Arrive By : 9:00am

SUMMARY OUTPUT :

Google Maps URL : <http://tinyurl.com/wg88j9q>
Total Distance : 23.9 mi
Total Duration : 1 hour 23 mins
Departure Time : 7:29am
Arrival Time : 8:52am

DIRECTIONS :

Trip Segment 1 : Walk to North Point Pkwy @ Encore Pkwy
Travel Mode : Walking
From : 1100 Somerby Ct, Alpharetta, GA 30009, USA
To : North Point Pkwy @ Encore Pkwy, Alpharetta, GA 30022, USA
Google Maps URL : <http://tinyurl.com/wzr4mon>
Distance : 0.8 mi
Duration : 15 mins
Departure Time : 7:29am
Arrival Time : 7:44am

Trip Segment 2 : Bus towards 141 North Springs Station Via Deerfield Webb Rd
Travel Mode : Transit
From : North Point Pkwy @ Encore Pkwy
To : North Springs Station
Google Maps URL : <http://tinyurl.com/umaqvww>
Distance : 8.5 mi
Duration : 20 mins
Departure Time : 7:44am
Arrival Time : 8:05am

Trip Segment 3 : Walk to North Springs Transit Station
Travel Mode : Walking
From : North Springs Transit Station, 7010 Peachtree Dunwoody Rd, Sandy Springs, GA 30328, USA
To : North Springs Transit Station, 7010 Peachtree Dunwoody Rd, Sandy Springs, GA 30328, USA
Google Maps URL : <http://tinyurl.com/qv437e5>
Distance : 7 ft
Duration : 1 min
Departure Time : 8:05am
Arrival Time : 8:05am

Trip Segment 4 : Metro rail towards Red Southbound to Airport Station
Travel Mode : Transit
From : North Springs Transit Station
To : North Avenue Marta Station
Google Maps URL : <http://tinyurl.com/vlymuhs>
Distance : 13.8 mi
Duration : 22 mins
Departure Time : 8:17am
Arrival Time : 8:39am

Trip Segment 5 : Walk to North Avenue MARTA Station
Travel Mode : Walking
From : North Avenue Marta Station, 713 Peachtree St NW, Atlanta, GA 30308, USA
To : W Peachtree St + Marta North Ave Station pm, Atlanta, GA 30308, USA
Google Maps URL : <http://tinyurl.com/qt4xsuy>
Distance : 180 ft
Duration : 1 min
Departure Time : 8:39am
Arrival Time : 8:39am

Trip Segment 6 : Bus towards 51 Boone Blvd - Holmes Stn - Luckie St
Travel Mode : Transit

From : North Avenue MARTA Station
To : Luckie St NW @ Merritts Ave
Google Maps URL : <http://tinyurl.com/tq4vepl>
Distance : 0.8 mi
Duration : 6 mins
Departure Time : 8:45am
Arrival Time : 8:51am

Trip Segment 7 : Walk to 528 Luckie St NW, Atlanta, GA 30313, USA
Travel Mode : Walking
From : Luckie St NW @ Merritts Ave, Atlanta, GA 30313, USA
To : 528 Luckie St NW, Atlanta, GA 30313, USA
Google Maps URL : <http://tinyurl.com/r242pvr>
Distance : 266 ft
Duration : 1 min
Departure Time : 8:51am
Arrival Time : 8:52am

Thank you for using this Custom Commute Planner v 1.0!