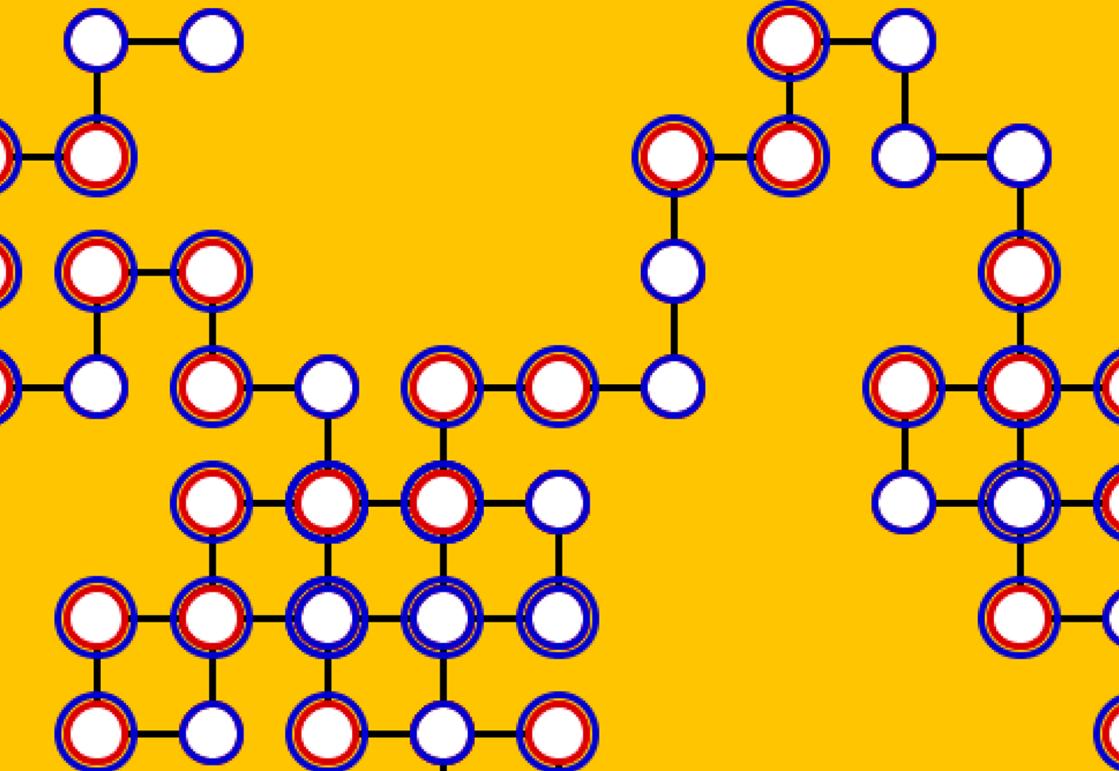


egGTFS 入門ガイド

飯倉宏治



egGTFS 入門ガイド

[著] 飯倉宏治

GTFS 活用ソフトウェアの開発に関する研究

2024 年 3 月 29 日 ver 1.0

■免責

本書は情報の提供のみを目的としています。

本書の内容を実行・適用・運用したことで何が起きようとも、それは実行・適用・運用した人自身の責任であり、著者や関係者はいかなる責任も負いません。

■商標

本書に登場するシステム名や製品名は、関係各社の商標または登録商標です。

また本書では、TM、^(R)、^(C)などのマークは省略しています。

まえがき

バスや電車の時刻表を取り扱うデータフォーマットに GTFS と呼ばれるものがあります。Google 社により策定されたこのフォーマットは広く一般的に利用されており、今や公共交通に関する標準的なデータフォーマットになっています。本書では、日本版の GTFS である GTFS-JP を簡単に取り扱うための Python 用ソフトウェア部品 egGTFS の基本的な使い方を説明しています。

Python には地図情報を扱うソフトウェア部品もあります。egGTFS とこれらを使うことで、特定の条件に合うバス停を地図上に表示するなど、様々な情報を簡単に地図上に示すことができます。このように、バスに関する視覚的なコンテンツを制作する場合にも活用できます。

このガイドでは、GTFS-JP について簡単な説明を行い、egGTFS のインストールの方法、egGTFS におけるデータの取り扱いについての基本的な考え方を示しつつ、具体的な使い方をサンプルプログラムと共に示してゆきます。

なお、本文中に掲載されるプログラムは Python で書かれていますが、Python のプログラミングについての解説は含まれていません。もし Python に慣れ親しんでいない方は、他の書籍等で Python プログラミングの基本的な事柄を学んでから本書をお読みになっていただければ幸いです。

この本が、GTFS-JP データの効果的な読み込み・書き出し、そして地図上での視覚化を通じて読者の皆さんに利益をもたらすことを願っています。さらに、egGTFS が公共交通の発展のお役に立てるることを心から願っています。

egGTFS の作者より

目次

まえがき	i
第 1 章 はじめに	1
1.1 GTFS とは	1
1.2 拡張フォーマット GTFS-JP	2
1.3 egGTFS	2
第 2 章 GTFS-JP データの基本	5
2.1 CSV ファイル	5
2.2 GTFS-JP を構成するファイルの概要	6
第 3 章 egGTFS のインストール	9
3.1 GitHub からダウンロード	9
3.2 インストール	11
第 4 章 GTFS-JP データの読み取り	13
4.1 egGTFS の利用準備	13
4.2 GTFS-JP データのオープン	13
4.3 データ読み取りの例	14
4.4 dump メソッドの応用例	15
4.5 各要素の取得方法	16
4.5.1 SingleRecord	17
4.5.2 RecordSet	17
4.5.3 CSV ファイルによる情報提供の確認	19
4.6 具体的な情報取得の例	19
4.6.1 便情報の取得	19
4.6.2 停車するバス停の情報	20
4.6.3 バス停の名称や位置の取得	22

第 5 章 地図上への可視化	25
5.1 可視化の例	25
5.2 バス停の描画	27
5.2.1 描画の準備	27
5.2.2 指定した便のバス停を描画するプログラム	28
5.3 経路の描画	31
5.3.1 shape 情報の取得と描画	31
5.4 経路とバス停の描画	32
[コラム]drawShape の実装	34
第 6 章 フィルタ処理と保存	35
6.1 filter の使い方	36
6.1.1 GTFS-JP データへの保存	37
6.1.2 フィルタ用の関数	37
6.2 filter メソッドの応用	38
6.2.1 対象の stop_id を特定する	39
[コラム] リスト内包表記を用いる方法	40
6.2.2 stop_id から関連する trip_id を求める	40
6.2.3 trip_id から route_id を求める	41
6.2.4 対象の route_id に限定した routes 情報を作り保存する	42
[コラム] フィルタ処理のテスト	44
第 7 章 egGTFS の活用例	47
7.1 路線バスの利用可能範囲の概算	47
7.1.1 バス停の住所を求める	47
7.1.2 stop_id から市の名前を取得する	49
7.1.3 バス停を中心とする円形領域の和の面積を求める	51
[コラム] 度形式と DMS 形式	53
[コラム] 処理時間の高速化	55
7.2 経路長の計算	56
7.2.1 shape 情報を使わずに経路長を求めるプログラム	56
7.2.2 shape 情報を用いて経路長を求めるプログラム	59

7.3	指定したバス停における待ち時間の分析	61
7.3.1	stop_id から stop_times の情報を集める	62
7.3.2	停車時刻からバスがやってくる頻度を求める	63
付録 A	メタプログラミングと egGTFS	67
A.1	單一行の場合 (SingleRecord を用いる場合)	68
A.2	複数行の場合 (RecordSet を用いる場合)	69
付録 B	API リファレンス	71
B.1	egGTFS モジュールの関数	71
B.1.1	egGTFS.open(gtfsFilePath)	71
B.2	egGTFS クラス	71
B.2.1	クラスメソッド	71
B.2.2	インスタンスマソッド	72
B.3	Time クラス	73
B.3.1	クラスメソッド	73
B.3.2	インスタンスマソッド	73
B.4	TimeDelta クラス	74
B.5	TimeDiff クラス	74
B.6	AreaRect クラス	74
B.6.1	クラスメソッド	74
B.6.2	インスタンスマソッド	74
B.7	GTFS-JP 内の情報を表すクラス	75
B.7.1	agency, agency_jp, stops, routes, routes_jp, trips, office_jp, calendar, calendar_dates, fare_attributes, fare_rules, frequencies, transfers, feed_info, translations クラス	75
B.7.2	stop_times クラス	75
B.7.3	shapes クラス	76
あとがき		77

第 1 章

はじめに

1.1 GTFS とは

電車やバスなどの公共交通機関は、私達の日々の生活に欠かせない存在となっています。通勤や通学、毎日の買い物や旅行で訪れた観光地での移動手段など、様々な目的で利用されています。これら公共交通の運行状況はもちろんのこと、どの様に路線が整備されているのか等も私達の生活に大きな影響を与えます。また、公共交通が効率的に運営されることは渋滞の緩和や環境負荷の軽減、ひいては社会全体の持続可能性にも関係しています。

現代の公共交通は数多くの情報から構成されています。例えば路線バスを考えてみて下さい。数多くのバス停があり、それらに何度もバスが到着・出発しています。ひとつのバス停に着目してみても、分刻みで数多くの情報が紐づいています。このような多量の情報を人手で処理するのはとても大変です。

公共交通に関するこれらの情報をコンピュータにて処理するためには、各種の情報をデータとして蓄えなければなりません。しかし、鉄道会社やバス会社がそれぞれ独自にデータフォーマットを定め、その書式に基づいてデータファイルを制作・提供していると、情報の共有や利用に手間を要する状況となってしまします。

幸い、このような状況には陥ってはおらず、公共交通に関する情報をコンピュータにて取り扱う方法として GTFS なるものが存在しています。これは General Transit Feed Specification の略で、交通機関の運行スケジュールや路線情報、停留所の位置などを標準化したデータフォーマットです。GTFS を利用することにより、公共交通に関する情報の共有と活用が促進できます。

1.2 拡張フォーマット GTFS-JP

GTFS の最初のバージョンは 2006 年に米国の Google 社より発表されました。当初は Google Transit Feed Specification の略としての "GTFS" でしたが、2009 年に現在の General Transit Feed Specification の略称を意味することとされました（略字の最初の G が Google ではなく General を表すように変化しています）。

GTFS は公共交通に関するデータフォーマットとして広く知られています。しかし特定の国や地域に特有な状況全てに対応できるものではありません。例えば我が国においては、漢字で表された地名等の読み方は非常に重要です。日本版の GTFS には、これらの「よみがな」に関する情報は必須とすべきであり、このような状況を考慮し国土交通省は 2017 年に GTFS を拡張した GTFS-JP を策定し発表しました。これは**「標準的なバス情報フォーマット」**とも呼ばれています。この拡張フォーマットでは先程例に挙げた「よみがな」が必須化されるなど、日本国内の状況に応じた仕様となっています。2021 年には改定された第 3 版の仕様書が公開されており、本稿執筆時においては、この仕様書は <https://www.mlit.go.jp/sogoseisaku/transport/content/001419163.pdf> にて確認できます。なお本書では特に断りのない限りは GTFS-JP の仕様書として、この第 3 版を指すこととします。また、これから紹介する egGTFS は GTFS-JP を対象としたソフトウェアですので、GTFS-JP を単に GTFS と記す場合もあります。

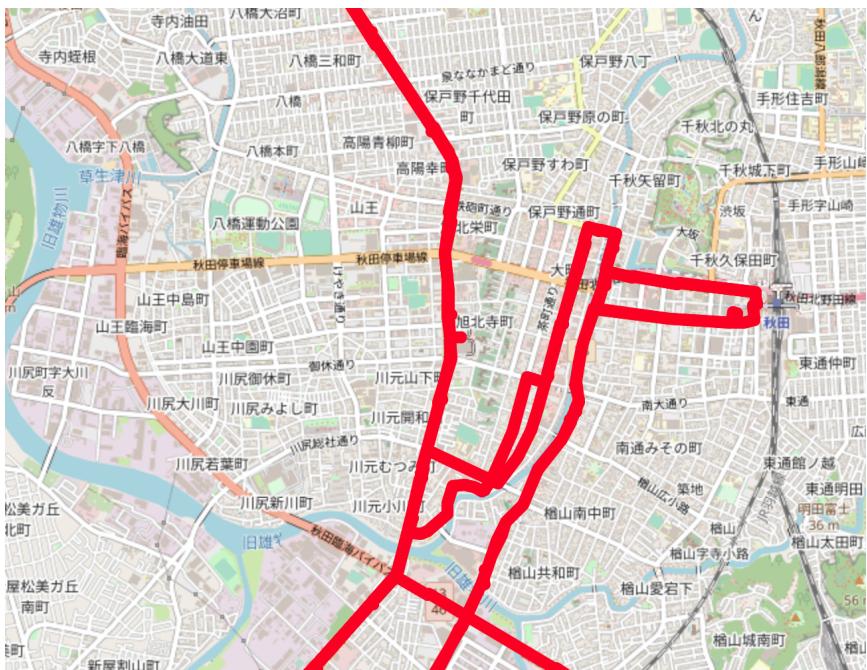
1.3 egGTFS

GTFS は、バス停の位置やバスの到着時刻や出発時刻など、様々なデータがまとめたものです。例えば、運行する便の情報は trips.txt という情報にまとめられています。これらの情報の実体は CSV 形式にて記述されたファイルです。複数の CSV 形式のテキストファイルを ZIP 形式にてひとまとめにしたもののが GTFS ファイルとなっています。

格納されている情報を取り出すためには ZIP ファイルを展開し、その後 CSV 形式のデータを読み取り…といった作業が必要となります。この処理は GTFS ファイルを取り扱う際には必ず発生します。これら、利用する場合に必要となる共通の手続きはソフトウェアライブラリとして提供されるべきです。

egGTFS はこのような共通の処理やバスに関する各種の情報処理機能をまとめた Python 用のソフトウェア部品です。egGTFS を用いると、先程示した trips.txt に格納されている情報は gtfs.trips[トリップ ID] という形で、指定したトリップ ID に関連する便情報を簡単に取得できます。

データの取得だけではなく、地図上へのデータの可視化などの機能も有しています。例えば、バス停の位置や路線を地図上に描画し、交通網の分析や最適化に役立てることが可能です。egGTFS はフィルタ機能も有していますので、ある条件を満たす情報のみを取り出すことも可能です。



▲ 図 1.1: とある条件を満たすバス路線の一部

egGTFS は複数の GTFS データを同時に利用できます。これにより他の地域の公共交通との比較なども容易に実現できます。公共交通の効率的な計画や運営、サービスの改善に携わる専門家にとって、有用なツールとなります。

第 2 章

GTFS-JP データの基本

GTFS-JP の詳細については 1.2 節で紹介した国土交通省にて公開されている仕様書に譲りますが、ここでは簡単に GTFS-JP について説明します。なお、既に述べているように GTFS-JP は GTFS を拡張したものですので、基本的な考え方は GTFS でも同じです。

2.1 CSV ファイル

1.3 でも言及していますが、GTFS も GTFS-JP も複数の CSV ファイルを ZIP 形式でまとめたものです。最初に CSV ファイルについて簡単に説明します。CSV とは Comma Separated Values の略で、テキストファイルにカンマ区切りでデータを記述したものを意味します。CSV ファイルに格納される情報は表形式のデータとなり、表の行にあたる部分は CSV ファイルでも改行記号と共に示される 1 行にて記述されます。

CSV ファイルは単なるテキストファイルですので、人間が読み書きすることができます。一方、それをどのように解釈するのかはその CSV ファイルがどのようなルールによって記されているかに依存します。特にプログラムで CSV ファイルを読み込む場合、注意しなければならないのは最初の行の取り扱いです。

CSV フィアルの最初の行はヘッダ行として、各列のタイトルを示す情報として取り扱われる場合があります。例えば、以下のような CSV ファイルの場合 sample.csv ファイルは全部で 4 行のテキストファイルですが、データとしては 3 行 2 列の表を現しています。ヘッダ行に各列のタイトルが示されており、第 1 列は商品名、第 2 列は価格というタイトルであると解釈できます。CSV ではヘッダ行は必須ではありませんが、GTFS ならびに GTFS-JP では

ヘッダ行のある CSV ファイルを取り扱います。

▼ sample.csv

```
商品名,価格
ミカン,100
リンゴ,200
バナナ,150
```

既に述べているように、CSV ファイルは単なるテキストファイルですので、文字コードや改行の表し方に気をつけなければなりません。GTFS-JPにおいてはその仕様書によると、文字コードは UTF-8 を、改行コードは CRLF または LF を用いることと規定されています（1-7-2. 利用可能文字等）。

2.2 GTFS-JP を構成するファイルの概要

GTFS-JP は様々な CSV 形式のファイルで構成されており、egGTFS ではこれらのファイル名を属性として使用します。そのため、これらのファイルの名前を把握することが重要です。以下に、GTFS-JP を構成するファイルを列举しておきます。

- **agency.txt** - 事業者情報 [必須]
- agency_jp.txt - 事業者追加情報 [任意]
- **stops.txt** - 停留所・標柱情報 [必須]
- **routes.txt** - 経路情報 [必須]
- **trips.txt** - 便情報 [必須]
- office_jp.txt - 営業所情報 [任意]
- pattern_jp.txt - 停車パターン情報 [任意]
- **stop_times.txt** - 通過時刻情報 [必須]
- **calendar.txt** - 運行区分情報 [条件付必須]
- **calendar_dates.txt** - 運行日情報 [条件付必須]
- **fare_attributes.txt** - 運賃属性情報 [必須]
- **fare_rules.txt** - 運賃定義情報 [条件付必須]

- shapes.txt - 描画情報 [任意]
- frequencies.txt - 運行間隔情報 [任意]
- transfers.txt - 乗換情報 [任意]
- **feed_info.txt** - 提供情報 [必須]
- **translations.txt** - 翻訳情報 [必須]

calendar.txt は仕様書によると「すべてのサービスの日付が calendar_dates.txt に定義されていない限り必須。」となっています。また、calendar_dates.txt は

calendar.txt が無い場合、本テーブルが必須となり全ての日について定義する必要がある。祝日等運行区分に基づかない例外的な運行をする日を設定。全ての不定期運行に対して設定することが望ましいが、設定が困難な場合は基本的な運行パターンを calendar で設定し、jp_trip_desc で例外がある旨を表示。

となっています。

fare_rules.txt については、「全線均一運賃の場合は不要、その他の場合は GTFSJP としては必須。」のことです。

第3章

egGTFS のインストール



Python の外部モジュールは、一般的に pip 等のコマンドを用いてインターネットから必要なファイルを自動的にダウンロードし、それらのファイルを利用してインストールされます。egGTFS の開発環境では pip を利用しているため、個人的には egGTFS においても pip install egGTFS としてインストールできることが理想的だと思っています。そのためには、PyPI に開発者として登録する必要があるのですが、本章執筆時においては、新規開発者登録を一時停止しているため、この方法は利用できません。そこで本章では GitHub からインストール用のパッケージをダウンロードし、pip を用いてインストールする方法を説明します。

3.1 GitHub からダウンロード

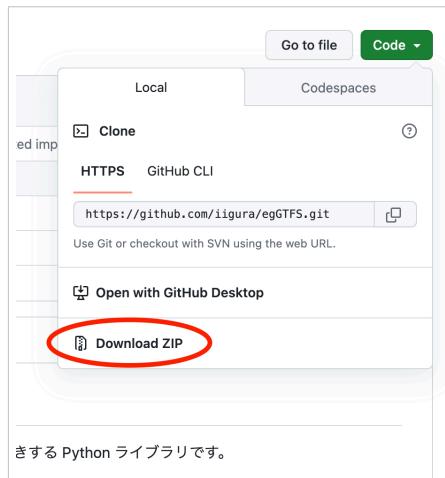
Web ブラウザより、<https://github.com/iigura/egGTFS> にアクセスすると、次のような画面が現れます。画面の右側の部分（分かりやすいように今回は赤丸を追加しています）に Code というボタンがありますので、それをクリックします。

ボタンをクリックすると "Download ZIP" というメニューが現れますので、それをクリックし egGTFS のモジュール一式をダウンロードします。このファイルを適当な場所に展開して下さい。

第3章 egGTFS のインストール

The screenshot shows the GitHub repository page for `iigura/egGTFS`. The main navigation bar at the top includes links for Product, Solutions, Open Source, Pricing, a search bar, and buttons for Sign in and Sign up. Below the header, the repository name `iigura/egGTFS` is displayed along with a Public status. A navigation bar below the repository name offers links to Code, Issues, Pull requests, Actions, Projects, Security, and Insights. The 'Code' link is highlighted with a red circle. On the left, there are buttons for main (1 branch), 1 branch, and 0 tags. The repository's activity feed shows several commits, with the most recent one being `Removed unrelated import statements.` by `fb1de20` 2 days ago. The commit count is 8. To the right of the code area, there is an 'About' section describing `GTFS-JP` as a Python library for reading GTFS files. It includes links to Readme, MIT license, Activity, stars (0), forks (0), and a Report repository button. Below the code area, there are sections for Releases (No releases published) and Packages. A note at the bottom states: "egGTFS は、GTFS-JP 形式のファイルを読み書きする Python ライブリです。"

▲図 3.1: <https://github.com/iigura/egGTFS>



▲図 3.2: ZIP ファイルにてダウンロード

3.2 インストール

無事に展開できたら、展開したファイルの中身を確認して下さい。

- LICENSE
- pkg
- README.md
- release_note.md
- samples

という内容になっていれば正常にダウンロードできています。なお、pkg と samples はディレクトリ（フォルダ）です。

これらのファイルが格納されているディレクトリを egGTFS-main であると仮定します（注：ディレクトリ名を egGTFS-main に変更する必要はありません）。コマンドプロンプトやターミナルを開き、そのディレクトリへ移動して下さい。

MacOS 等のいわゆる Unix 系の OS であれば ls を Windows であれば dir を実行すると、上記のファイルやディレクトリが表示されると思います。この状態にて、MacOS や Linux では

```
pip install ./pkg
```

を、Windows では

```
pip install .\pkg
```

と入力して下さい（実際の入力では円マークは半角記号で入力して下さい）。

無事にインストールが終了したら、念のため pip list と入力し、egGTFS がパッケージリストの一覧中に表示されていることを確認して下さい。

最後に python と入力し、Python の対話型環境を起動し、

```
>>> import egGTFS  
>>>
```

として egGTFS をインポートしてもエラーが発生しないことを確認して下さい。無事、インポートできれば egGTFS モジュールは正常にインストールされています。

第 4 章

GTFS-JP データの読み取り

4.1 egGTFS の利用準備

前の章の最後では、egGTFS が正常にインストールされていることを確認するために `import egGTFS` と Python インタプリタに入力していました。これはそのまま egGTFS の機能を Python プログラムで利用するために必要な処理となります。

egGTFS のインポート時にエラーが発生する場合は egGTFS が利用している各種のモジュール（プログラム部品）が不足している可能性がありますので、第 3 章を参考にしつつ必要なモジュールがインストールされているかを確認して下さい。

4.2 GTFS-JP データのオープン

egGTFS にて対象の GTFS ファイルを読み取るためには、egGTFS モジュール内の `open` 関数を呼び出します。例えば、`sampleGTFS.zip` という GTFS データを読み取るためには以下のようにします。

```
import egGTFS  
gtfs=egGTFS.open('sampleGTFS.zip')
```

以後、`sampleGTFS.zip` 内の情報を読み取る場合は、`open` 関数の戻り値を格納している `gtfs` という変数を用いることとなります。

4.3 データ読み取りの例

GTFS-JP により提供されているデータを活用する上で、データの読み取りは最も重要な処理のひとつです。ここからは実際に公開されている GTFS-JP データを用いて、具体的な読み取り方法について説明していきます。

それぞれのバス会社さんにて GTFS-JP データが公開されています。本書では <https://www.akita-bus.or.jp/pages/38/> にて公開されている秋田中央交通さんの GTFS-JP データを用いて説明を行っていきます。上記の Web ページから bus-akitachuoukotsu.zip をダウンロードして適当なディレクトリに保存して下さい。そのディレクトリにて Python の対話環境を起動し、次のように入力してみて下さい（以下は MacOS での例です）。

▼ agency.txt 情報の表示

```
>>> import egGTFS
>>> gtfs=egGTFS.open('bus-akitachuoukotsu.zip')
>>> gtfs.agency.dump()
agency_id      = 3410001000459
agency_name    = 秋田中央交通
agency_url     = https://www.akita-chuoukotsu.co.jp/
agency_timezone = Asia/Tokyo
agency_lang     = ja
agency_phone    = nan
agency_fare_url = nan
agency_email    = None
>>>
```

無事、agency.txt に格納されている情報が表示されたでしょうか？

このように egGTFS を使うと、簡単に GTFS-JP に格納されている情報を読み取ることができます。上のプログラムについて説明しますと、まず、egGTFS を import し、egGTFS.open にて GTFS-JP データを読み込んでいます。egGTFS.open 関数の戻り値は egGTFS クラスのインスタンスとなり、このインスタンスには GTFS-JP の情報が格納されています。GTFS-JP は複数の CSV 形式で記述されたテキストファイルを ZIP 形式でまとめたものでした。egGTFS クラスのインスタンスは、これらのテキストファイルのファイル名（厳密に言うと拡張子と呼ばれることがある末尾の .txt を含まない）

いファイル名) にて、それぞれのファイルに格納されている情報を取得することができます。

上に示した agency フィールドは、agency.txt に格納されている情報を保持しています。この agency フィールドは agency クラスのインスタンスです。これら各ファイルの情報を保持するためのクラスには、いくつかのメソッドが定義されています。dump メソッドはそのうちのひとつです。このメソッドは保持しているすべての内容を標準出力に表示するものです。例えば、バス停に関する情報は gtfs.stops に格納されていますので、gtfs.stops.dump() とすると、バス停の情報が全て表示されます。

4.4 dump メソッドの応用例

GTFS-JP は ZIP 形式でまとめられたものでした。そのため、それぞれの情報、例えば便名など便情報を確認するためには ZIP ファイルの中身をみて、便情報が格納されている CSV ファイル trips.txt の内容を表示させる必要があります。

今しがた紹介した dump メソッドを用いると、これらの内容は簡単に表示できます。次のプログラム dump.py を見て下さい。

▼ dump.py - GTFS-JP に含まれる CSV ファイルの内容表示

```
import sys
import egGTFS

if len(sys.argv)<3:
    print('usage: python dump.py GtfsFilePath csv-target')
    print('ex   : python dump.py yourGtfs.zip trips')
    exit(-1)

gtfs=egGTFS.open(sys.argv[1])
gtfs[sys.argv[2]].dump()
```

このプログラムは

```
python dump.py bus-akitachuoukotsu.zip trips
```

などとして使用します。dump.py の後には対象の GTFS-JP ファイルと表示したい CSV ファイルのファイル名（拡張子の .txt を含まず）を指定すれば、その CSV ファイルの中身を表示します。

最初の if 文は dump.py への引数の確認ならびに引数が不足していた場合の使い方の表示ですので、実際に表示処理を行うプログラム本体は

```
gtfs=egGTFS.open(sys.argv[1])
gtfs[sys.argv[2]].dump()
```

の 2 行のみです。

egGTFS のインスタンスは文字列で CSV ファイルを指定すればその情報を格納した egGTFS のフィールドが取得できる機能を有しており、dump.py はその機能を活用したプログラム例です。各行において表示される情報の順序は、GTFS-JP の仕様書に記載されている各 CSV ファイルの項目の順序と同じです。

このように egGTFS を用いると、簡単に GTFS-JP に格納された情報を取得できます。

4.5 各要素の取得方法

egGTFS では SingleRecord と RecordSet というクラスが存在します。これは egGTFS の用語であり、GTFS-JP の用語ではありません。GTFS-JP に格納されているファイルは、上で示したように egGTFS インスタンスのフィールドとして格納されています。これらのフィールドは SingleRecord または RecordSet というクラスより派生したものです。それぞれのクラスにより、各レコードの属性値の取得方法が異なります。

4.5.1 SingleRecord

agency.txt のように單一行（单一レコード）で構成されるファイルは SingleRecord クラスの派生クラスになっています。agency フィールドの他、agency_.jp や feed_info がこのクラスを継承しています。

これらのファイルの場合、特にレコードを指定する必要もなく、GTFS-JP にて定義されている各フィールド名にてそれぞれの情報を直接取得できます。例えば、feed_info.txt には feed_publisher_name というフィールドが定義されていますが、その情報を取得する場合は gtfs.feed_info.feed_publisher_name とすれば十分です。

▼ feed_info.txt 情報の表示

```
>>> import egGTFS
>>> gtfs=egGTFS.open('bus-akitachououkotsu.zip')
>>> gtfs.feed_info.feed_publisher_name
'秋田中央交通'
>>>
```

4.5.2 RecordSet

バス停のように、複数のレコードから成る CSV ファイルの情報は RecordSet クラスの派生クラスとして実装されています。GTFS-JP に格納されている CSV フィアルの情報は、上で示した SingleRecord の派生クラスで示されるもの以外は全て RecordSet クラスより派生しています。

レコードの指定方法

これらの情報では、インデックスとして利用できるフィールドを用いて特定のレコードを指定し、その上で各種のフィールド情報を取得します。例えば、バス停の情報を格納している stops.txt に関しては、バス停の ID を表す stop_id フィールドの情報がインデックス値として利用できるようになっています。そのため gtfs.stops[stop_id の値] とすれば指定したバス停の情報を得ることができます。

▼ 特定のバス停の情報を読み出す

```
>>> import egGTFS  
>>> gtfs=egGTFS.open('bus-akitachuoukotsu.zip')  
>>> gtfs.stops['9016_02'].stop_name  
'割山町'
```

上の例では、'9016_02'というstop_idを持つバス停の情報を取得し、そのstop_nameを表示しています。

なお、インデックスとして指定できる情報の種類は、次のとおりです：

CSV ファイル	インデックス値
calendar	service_id
calendar_dates	service_id
fare_attributes	fare_id
fare_rules	route_id
frequencies	trip_id
office_jp	office_id
routes	route_id
routes_jp	route_id
shapes	shape_id
stops	stop_id
stop_times	trip_id
transfers	from_stop_id
translations	trans_id
trips	trip_id

レコードからの情報取得

少し細かく説明すると、gtfs.stops['9016_02']にて取得できるのはstops_recordというクラスのインスタンスです。stopsに限らず、CSVファイルの拡張子を除いたファイル名に_recordという名前を付したもののが取得される単一のレコードを表します。これら～_recordというクラスのインスタンスはGTFS-JPで定義されている各種のフィールド名と同名のフィールドを有しています。そのため、stops[stop_id]により得られるレコードのstop_nameフィールドを指定することで、バス停の名前が取得できます。

イテレータ

特定のレコードを取得するのではなく、特定の CSV ファイルに存在する全てのレコードをひとつづつ取り出して処理を行いたい場合もあります。そのような用途に対応するため、RecordSet の派生クラスはイテレータとしても利用できるよう設計されています。

例として示している stops にも、もちろんこの機能は実装されており、次のようにすれば、全てのバス停の情報を表示できます：

▼ 全てのバス停の情報を表示する

```
import egGTFS
gtfs=egGTFS.open('bus-akitachoukotsu.zip')
for s in gtfs.stops: print(s)
```

4.5.3 CSV ファイルによる情報提供の確認

GTFS-JP に含まれている CSV ファイルについて、そのファイルが提供されているか否かは valid というフィールドにて取得できます。例えば agency が提供されているか否かは gtfs.agency.valid で調べることができます。これは SingleRecord と RecordSet、両方のクラスに実装されている機能です。

4.6 具体的な情報取得の例

本節では egGTFS を使用して GTFS データから具体的な情報を取得する方法を紹介します。これらは次章で説明する地図上での可視化において重要な役割を果たします。ここからはどのようにプログラムを作成していくのか、具体的な考え方も示しつつ解説していきます。

4.6.1 便情報の取得

今、「とある便に関する情報を抽出したい」、もしくは「抽出しなければならなくなってしまった」と仮定しましょう。GTFS の仕様書を読んでみると、便情報は trips.txt に格納されていることが分かります。このような場合は、まずは対象

となっている便に関する情報を調べなければなりません。そのため、先ほど示した dump.py 等を用いて対象の便の具体的な ID 値などを集めます。

先程示した表によれば trip_id により trips から具体的な便の情報（関連する CVS ファイルの行の情報）を得られそうです。実際、取得したい便の trip_id が' 平日_07 時 10 分_系統 71101 ' という値（文字列）であった場合は

```
import egGTFS
gtfs=egGTFS.open('bus-akitachuoukotsu.zip')
print(gtfs.trips['平日_07時10分_系統71101'])
```

とすれば、その便に含まれる trips 関連の情報を表示できます。

現在着目している便が、どのような経路を運行するのかを表す route_id も以下のように取得できます。

```
>>> gtfs.trips['平日_07時10分_系統71101'].route_id
'新屋線_A'
>>>
```

ここで表示されている ' 新屋線_A ' という値が現在注目している便の route_id となり、経路に関する情報を取得する際には重要な情報となります。

また、地図上に経路を表示する場合においては route_id ではなく shape_id が必要になります。こちらも便情報に含まれていますので、route_id と同様に

```
>>> gtfs.trips['平日_07時10分_系統71101'].shape_id
71101
>>>
```

などと簡単に取得することができます。なお、地図上に経路を描画するために必要となる shapes に関しては次の章にて詳しく説明します。

4.6.2 停車するバス停の情報

便情報から路線や路線の経路についての情報を取得できることが分かりました。次は特定の便において、どのバス停に停車するのか、それらの情報を取得してみましょう。

バス停の停車情報については通過時刻情報である stop_times に格納されています。stop_times は trip_id を用いて取得できますので、'平日_07 時 10 分_系統 71101' という trip_id に関する stop_times は gtfs.stop_times['平日_07 時 10 分_系統 71101'] として得ることができます。

このようにして得た通過時刻情報は複数のバス停の情報を含むため gtfs.stop_times['平日_07 時 10 分_系統 71101'] といった形で取得できる情報は配列となります。なお、配列に格納される順序は stop_times_record に格納されている stop_sequence の値順になっています。

GTFS の仕様書によると stop_sequence は必ずしも連番である必要はないとのことです。egGTFS では stop_sequence の値が昇順になるよう stop_sequence の配列に格納していますので、例え GTFS に格納されている stop_sequence の値が連番でなくとも配列の要素に None などの空要素が含まれることはありません。そのため得られた配列要素を順次取得していくば、それだけで当該の便のバスが停車するバス停を順次列挙できます。

次のプログラムはある便に関する stop_sequence と stop_id を示すものです：

▼ stop_id が昇順になっている様子

```
>>> for t in gtfs.stop_times['平日_07時10分_系統71101']:
...     print(t.stop_sequence,t.stop_id)
...
1 1022_07
2 3160_02
3 2174_02
(中略)
27 1059_01
28 4029_01
29 1060_01
```

左側の数字が stop_sequence の値ですが、特になにもしなくとも昇順になっている様子が確認できます。右側はバス停を示す stop_id の値です。次はこれらの stop_id からバス停の名前などを取得するプログラムを考えてみましょう。

4.6.3 バス停の名称や位置の取得

上で示したスクリプトにてバス停を識別する stop_id が取得できているので、これを使ってバス停の情報を取得すればよさそうです。少しプログラムが複雑になってくるので、今度はインタプリタへの入力ではなく、ファイルに保存するスクリプトとして記述してみます。

▼ 指定した便が停車するバス停を順番に表示するプログラム

```
import egGTFS

gtfs=egGTFS.open('bus-akitachuoukotsu.zip')
targetTripID='平日_07時10分_系統71101'
for t in gtfs.stop_times[targetTripID]:
    stopName=gtfs.stops[t.stop_id].stop_name
    print(t.stop_sequence,stopName)
```

このプログラムを実行すると、次のような表示が得られます：

▼ 停車するバス停を表示した様子

```
1 秋田駅西口
2 千秋公園入口
3 木内前
:
( 中略 )
:
27 新屋高校入口
28 田尻沢東町
29 新屋高校前
```

秋田中央交通さんでは、停車順序である stop_sequence をきちんと 1 つづつ増加させているのでこのような表示となりますが、左側に表示される番号は昇順になっているだけであり、必ずしもこのように表示されるわけではありません。

次の可視化の章では、バス停を地図上に描くことを試みます。そのため、ここでバス停の位置の表示についても説明しておきましょう。バス停の位置は緯度と経度により示されます。それぞれ stops レコードの stop_lat と stop_lon 列にて提供されています。こちらも egGTFS であれば単にそれぞれをフィールド名として指定すれば取得できますので、バス停の名前とそのバ

ス停が置かれている緯度と経度を表示するプログラムは次のように記述できます。

今回のプログラムは stops レコードの stop_sequence の昇順の様子に左右されないよう、Python の enumerate にてその順序を取得するように変更しています。これにより、何番目に停車するバス停なのかを正確に表示することができるようになっています：

▼ 指定された便が停車するバス停の名前と位置の表示

```
import egGTFS

gtfs=egGTFS.open('bus-akitachoukotsu.zip')
targetTripID='平日_07時10分_系統71101'
for index,t in enumerate(gtfs.stop_times[targetTripID]):
    stop=gtfs.stops[t.stop_id]
    stopName=stop.stop_name
    print(index+1,stopName,stop.stop_lat,stop.stop_lon)
```

上記のプログラムの実行結果は次のようになります：

```
1 秋田駅西口 39.7171895832309 140.128297805786
2 千秋公園入口 39.7178497740541 140.12327671051
3 木内前 39.7181469660994 140.120315551758
:
( 中略 )
:
27 新屋高校入口 39.6726106374828 140.096840858459
28 田尻沢東町 39.6706616196933 140.096840858459
29 新屋高校前 39.6673582391174 140.096368789673
```

左から停車順序、停車するバス停の名前、そのバス停の緯度と経度です。

このように egGTFS を用いると、簡単に GTFS-JP の情報が取得できます。

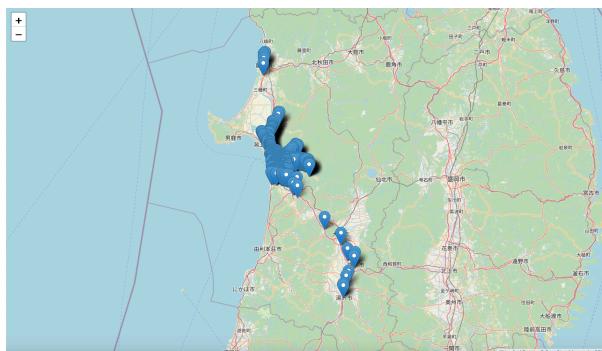
第 5 章

地図上への可視化

5.1 可視化の例

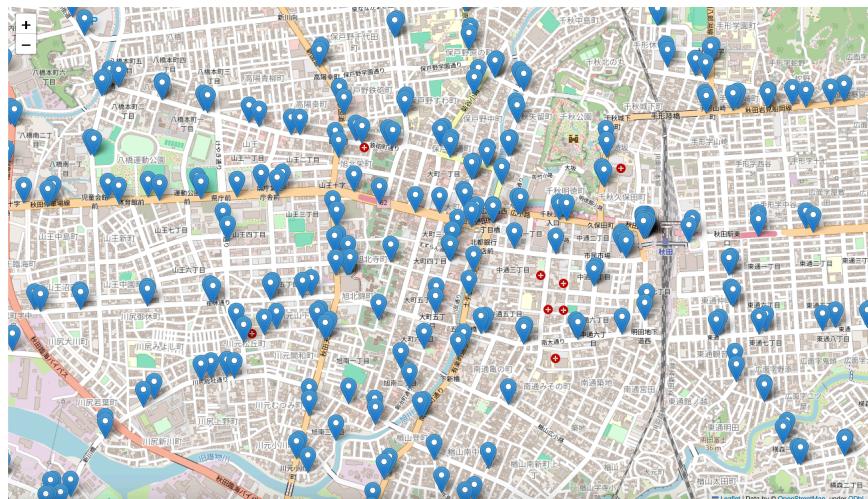
egGTFS では folium という地図表示モジュールを利用し、地図上にバス路線等を可視化することができます。egGTFS のメソッドでは、この folium の地図オブジェクト (Map オブジェクト) をそのまま返すものも存在します。例えば getAllStopsMap メソッドでは、GTFS-JP に格納されているバス停を描画した Map オブジェクトを返します。

Map オブジェクトには save メソッドが存在しています。このメソッドにて HTML ファイルとして保存し、そのファイルをブラウザにて表示すれば、該当の GTFS-JP に含まれる全てのバス停が確認できます。



▲ 図 5.1: 全てのバス停を地図上に表示した様子

この getAllStopsMap では、全てのバス停を含む表示となるように地図の拡大率が設定されています。そのため、バスの運行が広範囲に渡る場合は少々見づらい地図として表示されてしまうかもしれません。しかし、folium の地図は Google Map と同様にスクロールや拡大縮小ができますので、より詳しく確認したい場所を表示させることも可能です。



▲ 図 5.2: 秋田駅付近のバス停の様子

getAllStopsMap メソッドを用いた可視化の方法は、以下のようなプログラムにて簡単に実現できます。

▼ getAllStopsMap 関数の使用例

```
import egGTFS

gtfs=egGTFS.open(対象の GTFS-JP ファイルへのパスを文字列にて指定)
m=gtfs.getAllStopsMap()
m.save('ex_AllBusStops.html')
print('done: ex_AllBusStops.html is generated.')
```

生成された HTML ファイル ex_ALLBusStops.html をブラウザで開けば上で示したような表示が得られます。

上で示したプログラムは ex_makeAllBusStopsHTML.py として samples ディレクトリに同梱されていますので、皆さんの環境でも試してみて下さい。サンプルプログラムでは秋田中央交通さんの GTFS-JP ファイルを指定していますが、実際に動作させる場合には皆さんのが用意した GTFS-JP ファイルへのパスを指定するように変更の上、実行して下さい。

5.2 バス停の描画

ここからは、指定したとある便の運行ルートや停車するバス停を地図上に描画し、当該の便の可視化を行ってみようと思います。GTFS-JP では運行ルートの情報は内包される shapes.txt にて提供されます。しかし、この情報は任意の項目であるため、まずは必須情報であるバス停の位置を地図上に描画するところから始めていきたいと思います。

5.2.1 描画の準備

地図への描画は folium という Python のモジュールを用いて行います。そのため、

```
import folium  
m=folium.Map()
```

として、folium の Map オブジェクトを生成しておきます。

folium では Marker という関数を使って地図上にマーカーを設置できます。詳しくは folium の解説書に譲りますが、

```
folium.Marker(location=マーカーの位置,  
              popup=クリックしたときの表示).add_to(Map オブジェクト)
```

とすれば Map オブジェクトが表す地図にマーカーを設定できます。

今回はマーカーでバス停の位置を示しますので、そのマーカーをクリックしたらバス停の名前を表示しようと思います。folium の場合、popup に直接バス停の名前を指定すると、意図しないところで改行されてしまいます。このよ

うな現象を回避するためには、HTML の span タグにて style="white-space: nowrap;" なるスタイル指定をしなければならないのですが、これを毎回プログラムで書くのも興醒めです。

このような状況を考慮し、egGTFS では folium のマーカー用に、上記のスタイル指定を含んだ span タグにて、与えられた文字列を囲んだものを返すメソッドを用意しています。それが `makeName` です。

5.2.2 指定した便のバス停を描画するプログラム

前の章で、指定した便が停車するバス停の位置と名前を取得する方法を示しました。それを活用して、停車するバス停を地図上に描画するプログラムを作ってみたいと思います。

描画の様子は Web ブラウザを用いて確認するため、表示に用いる HTML ファイルを生成しなければなりません。既に示しているとおりですが、folium の Map オブジェクトには `save` メソッドが存在します。`m.save('test.html')` 等とすれば Map オブジェクト `m` に格納されている地図を表す HTML ファイルを生成できます。

しかし、ここで注意しなければならないのは、folium の Map オブジェクトに対し単に `save` オブジェクトを呼び出すと世界地図全体が表示される状況となってしまいます。繰り返しになりますが、folium で生成される HTML ファイルは Google Maps のようにマウスで地図を移動させたり拡大縮小できる機能を備えています。そのため、何も工夫をせずに `save` メソッドを呼び出し、毎回興味のある領域まで拡大する操作を行っても良いのですが、どうせなら最初から様子がわかる状態で表示できるようにしたいものです。

このような要求に対応するために、Map オブジェクトの `fit_bounds` メソッドを呼び出しておきます。これは、デフォルトで表示する地図の領域を緯度経度の矩形で指定するメソッドです。今回の場合はバス停が設置されている緯度と経度のそれぞれの最大値と最小値を `maxLat`, `maxLon` および `minLat`, `minLon` に保存しておく、それを `fit_bounds` メソッドの引数として活用すれば良いでしょう。これらを考慮して作成したプログラムを以下に示します：

▼ 指定した便に関するバス停を表示するプログラム

```
import egGTFS
import folium

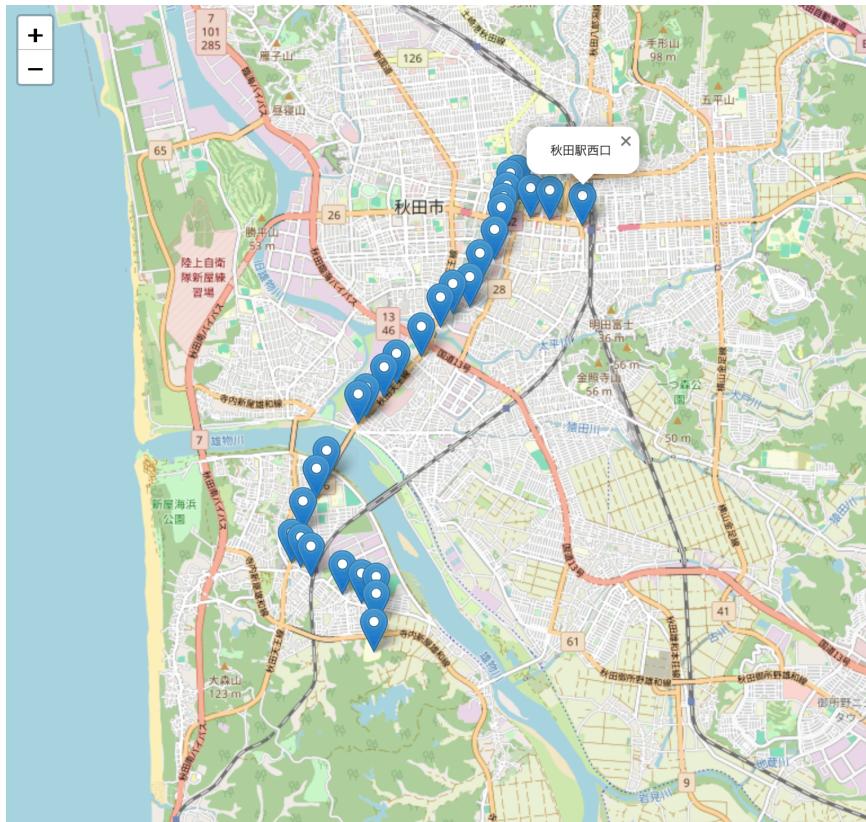
gtfs=egGTFS.open('bus-akitachoukotsu.zip')
m=folium.Map()

targetTripID='平日_07時10分_系統71101'
minLat=minLon=180
maxLat=maxLon=0
for index,t in enumerate(gtfs.stop_times[targetTripID]):
    stop=gtfs.stops[t.stop_id]
    stopName=stop.stop_name
    lat,lon=stop.stop_lat,stop.stop_lon
    folium.Marker(location=[lat,lon],
                  popup=gtfs.makeName(stopName)).add_to(m)
    minLat=min(minLat,lat); maxLat=max(maxLat,lat)
    minLon=min(minLon,lon); maxLon=max(maxLon,lon)
m.fit_bounds([[minLat,minLon],[maxLat,maxLon]])
m.save('test.html')
```

上記プログラムを実行すると test.html が生成されますのでそれをブラウザで開くと次のようになります。

この図では「秋田駅西口」という表示があります。これは、当該のマーカーをクリックした状態にてスクリーンショットを撮っているのでこのような画像になっています。プログラム中では stopName 変数にそれぞれのバス停の名前が格納されており、それを Marker を配置する際、popup 引数に指定しているため、このような挙動になっています。

上で示したプログラムは汎用的なものとして作成しています。そのため、targetTripID 変数を変更することにより、読者の皆さんのが興味を持っている便に関するバス停を地図上に表示することが可能です。興味のある方は、targetTripID の値をご自身の住まわれている地域のバスの便に変更して試してみて下さい。



▲図 5.3: 指定した便が停車するバス停の様子

5.3 経路の描画

提供されている GTFS-JP ファイルによっては、経路の情報を格納する shape.txt が同梱されているものもあります。これはバスがどのような経路を運行するのか、その地図上の形状を表した情報となります。形状と言っても実際には線分の集合で構成された折れ線です。

folium の機能を用いて、これらの線分をひとつひとつ描画することももちろんできます。しかし egGTFS には drawShape という経路を描画する専用のメソッドが用意されています。このメソッドを利用すれば、これらの処理ルーチンを皆さんができる必要はありません。

5.3.1 shape 情報の取得と描画

経路を描くためには shape 情報である shape_id を取得しなければなりません。これはそれぞれの便情報に格納されていますので、

```
gtfs.trips[tripID].shape_id
```

として取得できます。なお、この例では変数 gtfs に GTFS-JP の情報が格納されているものとしています。

drawShape メソッドを呼び出す際には描画先の folium の Map インスタンスと描画したい shape_id を指定して下さい。具体的な呼び出し方法は次のようにになります：

```
gtfs.drawShape(mapObject,shapeID)
```

描画する際には色や線の太さを指定したい場合もあります。そのような場合は、

```
gtfs.drawShape(mapObject,shapeID,width=線の太さ,color=色)
```

として drawShape メソッドを呼び出して下さい（デフォルト値は width=8, color="#FF0000" です）。色の指定方法は RGB 順です。color で与えられ

た値は folium の PolyLine メソッドに渡されます。そのため描画色に関する指定方法について、詳しくは folium のマニュアル等を参照して下さい。

5.4 経路とバス停の描画

以上を組み合わせて経路とバス停を地図上に描画してみます。実際に動作するプログラムは以下のようになります：

▼ 指定した便の経路とバス停の様子

```
import egTFS
import folium

gtfs=egTFS.open('bus-akitachoukotsu.zip')
m=folium.Map()

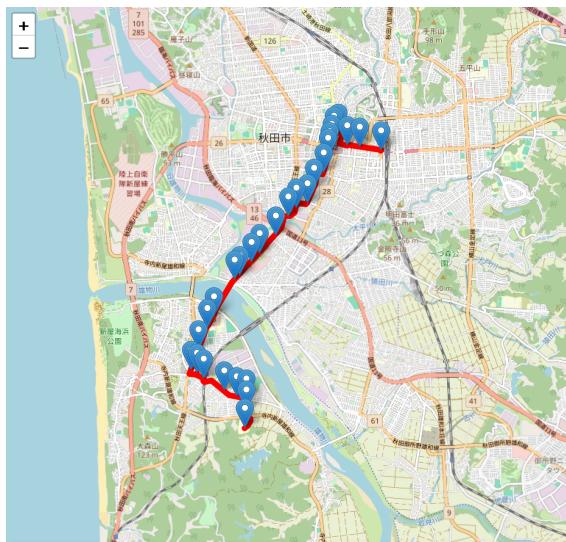
targetTripID='平日_07時10分_系統71101'

# 経路の描画
shapeID=gtfs.trips[targetTripID].shape_id
gtfs.drawShape(m, shapeID)

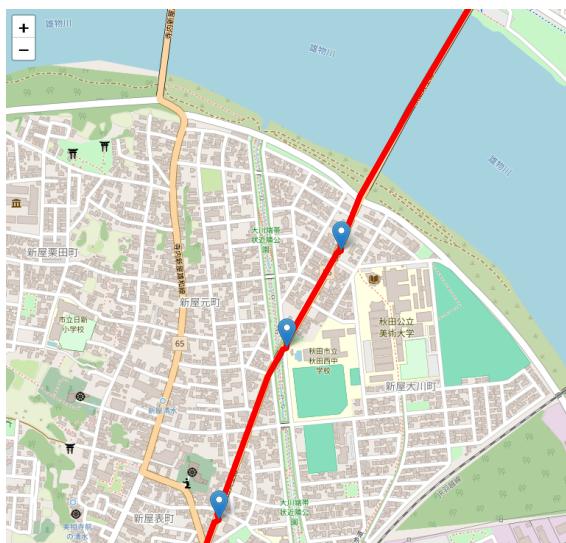
# バス停をマーカーで配置
minLat=minLon=180
maxLat=maxLon=0
for index,t in enumerate(gtfs.stop_times[targetTripID]):
    stop=gtfs.stops[t.stop_id]
    stopName=stop.stop_name
    lat,lon=stop.stop_lat,stop.stop_lon
    folium.Marker(location=[lat,lon],
                  popup=gtfs.makeName(stopName)).add_to(m)
    minLat=min(minLat,lat); maxLat=max(maxLat,lat)
    minLon=min(minLon,lon); maxLon=max(maxLon,lon)
m.fit_bounds([[minLat,minLon],[maxLat,maxLon]])
m.save('pathAndStops.html')
```

なお、drawShape の戻り値は指定された shape_id を描画した地図上の描画範囲を示す AreaRect クラスのインスタンスとなります。AreaRect クラスについては付録に説明がありますので、詳しくはそちらを御覧ください。

生成された地図は以下のようになります。



地図を拡大した様子：



【コラム】drawShape の実装

本文中でも示したように、shape.txt に格納されている線分情報を利用して折れ線としてひとつひとつ線分を描画していくても良いのですが、folium には折れ線を描く PolyLine メソッドが備わっています。drawShape メソッドでは、経路情報を PolyLine 用の情報に変換し、それを用いて PolyLine メソッドにてまとめて描画するという方法をとっています。参考までに、実際のソースコードを以下に示しておきます。

▼ drawShape の実際のコード

```
def drawShape(self,inMap,inShapeID,weight=8,color="#FF0000"):
    shapeArray=self.shapes.getShapeArray(inShapeID)
    points=[]
    latMin,latMax=360,0
    lonMin,lonMax=360,0
    for s in shapeArray:
        pos=self.shapes.pos(s)
        points.append(pos)
        latMin,latMax=min(latMin,pos[0]),max(latMax,pos[0])
        lonMin,lonMax=min(lonMin,pos[1]),max(lonMax,pos[1])
    w=weight
    c=color
    folium.PolyLine(points,weight=w,color=c).add_to(inMap)
    area=AreaRect()
    area.minLat=latMin
    area.maxLat=latMax
    area.minLon=lonMin
    area.maxLon=lonMax
    return area
```

第 6 章

フィルタ処理と保存

egGTFS では ver.2.0 よりフィルタ処理ならびに GTFS-JP への書き出しに対応しました(書き出しメソッドは `save` という名前で、`gtfs.save('newGtfsJp.zip')` 等として使用します)。これにより、特定の条件を満たす情報を探す部分と、そうでない部分の処理の分離が可能となります。

例えば、条件に合致する便情報を抽出し、その便のみから構成される GTFS-JP ファイルを出力するプログラムと、指定された GTFS-JP に含まれる便の経路を全て描画するプログラム等といった使い方です。

処理を分割せずひとつのプログラムで一括して処理する場合は、どのような条件においても地図上への描画処理は共通となります。抽出条件が変化すれば別のプログラムを作成しなければなりませんが、その際、描画部分は既存の描画処理のプログラムをコピー＆ペースト（いわゆる「コピペ」）にて記述するか、場合によっては描画部分をソフトウェア部品化して利用することになると思います。

描画処理をソフトウェア部品として部品化する場合は問題ありませんが、コピペの場合はソフトウェアの保守管理の観点から考えるとあまりよろしくありません。というのもコピペの場合、複数の場所に同様のプログラムが分散して保存されることとなります。このような状況の下、もし描画部分に修正を施す必要が出てきた場合はコピーされた全てのプログラムに対し、必要な修正を施さなければならなくなってしまいます。これは作成済みのソースコードの内容を全て把握しておく必要がありますし、修正漏れの可能性もありますので、かなり手間のかかる作業となってしまいます。

一方、情報の抽出と描画が分離されている場合は、全ての描画処理を担うプログラムはひとつだけとなりますので、修正作業は最小限となります。

もちろんフィルタ処理や GTFS-JP 形式での書き出し機能はこのような用途に限ったものではありません。様々な用途に利用可能な機能です。しかし、上で示した例は使い方として理解しやすく、実用的です。ここからはフィルタ処理と GTFS-JP 形式への保存機能の応用例として、実際にプログラムを作つてみたいと思います。

6.1 filter の使い方

stops など、複数の情報を保持するフィールドについては、filter メソッドが用意されています。例えばある集合 targetRouteIDs に含まれている特定の経路のみからなる GTFS-JP データを作りたいのであれば

```
gtfs.routes.filter(lambda gtfs, route: route.route_id in targetRouteIDs)
```

とすれば十分です。filter メソッドは

```
filter(フィルタ関数)
```

として使います。デフォルトではフィルタした結果を GTFS-JP データに反映するようになっています。もし、保持する GTFS-JP データに反映して欲しくない場合は

```
filter(フィルタ関数, update=False)
```

として呼び出します（引数 update のデフォルト値は True として設定されています）。

もし残したい経路 ID が分かっているのであればそれらから成る集合を targetRouteIDs 変数に格納しておき、

```
gtfs.routes.filter(lambda gtfs, route: route.route_id in targetRouteIDs)
gtfs.save('filteredGtfsJP.zip')
```

とすれば、指定した経路のみを含む GTFS-JP データが生成されます。

参考：filter 実行後の事業者情報

混乱を避けるため filter メソッド使用後は、事業者情報 (agency.txt) が egGTFS にて生成したことを示すものに置き換わります。

今回与えた lambda 式では、関数本体では使用していない gtfs という引数が存在していますが、これについては後ほど説明します。

6.1.1 GTFS-JP データへの保存

なお、save メソッドは egGTFS のインスタンスが持つメソッドです。引数に出力したい GTFS-JP のファイル名（ファイルパス）を指定すれば、保持しているデータを GTFS-JP フォーマットで出力します。ファイル名には .zip の拡張子まで含めるようにして下さい（save メソッドでは .zip の拡張子は自動的に付与しません）。

6.1.2 フィルタ用の関数

filter メソッドに与えるフィルタ用の関数は、残したい情報に対して True を返すようにして下さい。なお本書では、このような関数を述語と呼ぶ場合もあります。

フィルタ用の関数は egGTFS のインスタンスと対象のレコードを引数として受け取るようにして下さい。これは filter 関数内で egGTFS の情報を活用したい場合に備えるため、このような仕様になっています。

先程の例では lambda 式で関数を与えていましたが、フィルタ用の関数を別途定義するのであれば、

```
def someFunc(gtfs, record):
    :
    関数本体の定義（残したい情報ならば True を返す）
    :
```

となります。

第 2 引数で示されている record は、filter メソッドを実行するオブジェクト

ト毎に異なります。stops フィールドに対し filter メソッドを実行する場合は stops.txt の各行を示すオブジェクトになりますし、trips に対して filter メソッドを実行するのであれば変数 record には trips.txt の各行を示すオブジェクトとなります。

6.2 filter メソッドの応用

先程示した例では、フィルタ後に残したい経路 ID が分かっているという前提で説明をしました。そうではなく、そもそも残したい経路 ID 自体を調査して取り出したい、という場合はどうすればよいのでしょうか？

既に説明しているように egGTFS では GTFS-JP の ZIP ファイル内の複数のレコードを含む各ファイルに対応するフィールドが、ファイル名と同名で設定されています。これらのフィールドはイテレータの機能を有しており、データの反復処理を容易に実行できます（コラムを参照）。

filter 関数はその性質上、対象のフィールドが含む（つまり対象の CSV が含む）全ての情報を順次列挙し確認します。この性質を活用すると、プログラム中にてプログラマがループ処理を記述する必要が無くなります。

ここでは、とあるバス停に停車する経路情報のみを取り出し、GTFS-JP データへと保存するプログラムを考えてみたく思います。天下り的となり恐縮ですが、この処理は次の手順にて実現できそうです：

1. 指定された stop_id を含む通過時刻情報から関連する trip_id を収集
2. 収集した trip_id を含む便情報から route_id を収集
3. 収集した route_id のみをフィルタ

対象とするバス停を決めなければなりませんが、ここでは筆者の勤務先である美術大学前のバス停を例としてみましょう。

6.2.1 対象の stop_id を特定する

まずは、バス停の正確な stop_id を求めなければなりません。おそらく「美術大学前」というバス停だったと記憶していますが、念のため、それも調べてみましょう。調べる方法としては「美術」という文字列を含むバス停を全て収集し、表示してみます（プログラムの解説は後ほど行います）。

▼ バス停の名前に「美術」を含むものを表示

```
import egGTFS
srcGtfsFilePath='bus-akitachoukotsu.zip'
gtfs=egGTFS.open(srcGtfsFilePath)
def searchFilter(inGtfs,inStop):
    if '美術' in inStop.stop_name:
        if inStop.stop_id not in targetBusStopIDs:
            print('ID=',inStop.stop_id,'name=',inStop.stop_name)
    return False
gtfs.stops.filter(searchFilter,update=False)
```

このプログラムを実行すると、

```
ID= 6043_01 name= 美術大学前
ID= 6043_02 name= 美術大学前
```

という表示が得られます。バス停の名前も同じなので、おそらく（いわゆる）上りと下りの 2 つのバス停が存在し、それらが列挙されているのでしょう。

このプログラムでは何が行われているのでしょうか？ 最後の行をみてみると stops に対して filter メソッドが実行されてるのが分かります。そしてその filter メソッドの第 1 引数は searchFilter となっています。

searchFilter は filter メソッドから呼び出される述語として機能するため、def searchFilter(inGtfs,inStop) と宣言されています。最初に egGTFS のインスタンスを受取り、第 2 引数として stops の各レコードの情報を含むオブジェクト（実態は stops_record）を受け取っています。

stops クラスに対し呼び出される filter メソッドは、引数として渡された関数（今回の例では searchFilter 関数）を、stops の各行に対して呼び出します。そして filter メソッドから呼び出された searchFilter 関数では、第 2 引

数に stops.txt に格納されているそれぞれのバス停の情報からバス停の名前 (stop_name) を参照し、「美術」という文字列が含まれていればそのバス停の情報を表示しています。

filter メソッドから呼び出される関数は、True か False を返す必要がありました。今回の searchFilter 関数では最後に常に False を返していますが、filter メソッドの呼び出し時に update=False としていますので、searchFilter 関数が True を返しても False を返しても、gtfs 変数に格納されている情報には何も変化はありません（ですので今回の例では searchFilter 関数の戻り地には意味はありません。しかし、filter メソッドの仕様により、真偽値は返さなければなりません）。

【コラム】リスト内包表記を用いる方法

stops についてもイテレータの機能を有しているため、条件に合致する stop_id を集めるのであればリスト内包表記を用いて同様の処理も可能です。

```
[ stop.stop_id for stop in gtfs.stops if '美術' in stop.stop_name ]
```

プログラムを書く人はもちろんのこと、読む可能性の有る人達全員がリスト内包表記に慣れている場合は、こちらの記述の方が良いかもしれません。

6.2.2 stop_id から関連する trip_id を求める

この例では stop_id から route_id を求めなければなりません。stop_id と route_id を直接紐づける情報があれば良いのですが、残念ながらそのような直接的な情報は提供されていません。ここでは trip_id を経由して stop_id から route_id を求めるプログラムを作ってみます。

対象のバス停を識別する stop_id が分かったら、次はそのバス停を通過する便の trip_id を求めます。求め方ですが、stop_times.txt の各行の情報を見てみると、stop_id と trip_id が含まれていることが分かります。今回はこれを活用します。

stop_times の各行に対し、その行の stop_id と、対象のバス停の stop_id を比較して、等しい場合はその行に含まれる trip_id をリストに記録するようにします。もちろん、単にリストに追加してしまうと、重複して追加されてしまうので、それも考慮してプログラムを書いてみます。

▼ targetTripIDs に trip_id を集める

```
targetTripIDs=[]
def stopTimesFilter(inGtfs,inStopTime):
    if inStopTime.stop_id in targetBusStopIDs:
        if inStopTime.trip_id not in targetTripIDs:
            targetTripIDs.append(inStopTime.trip_id)
    return False
gtfs.stop_times.filter(stopTimesFilter,update=False)
```

このプログラムでは、対象とするバス停の stop_id は targetBusStopIDs というリストに格納されているものとします。上記のプログラムの実行後は targetTripIDs に対象とする trip_id の値が集められます。2 つ目の if 文は 対象のバス停に停車する便の ID (=trip_id) が既に targetTripIDs に含まれているか否かを調べており、もし含まれていなかったら targetTripIDs リストに追加する - そのような処理を行っています。

6.2.3 trip_id から route_id を求める

targetTripIDs に対象の trip_id を収集できたので、今度はこれら trip_id の値から route_id を求めてみましょう。trip_id と route_id は trips.txt の各行にて紐づけされているので、trips.filter() を用いて対象の trip_id に関する route_id を求めてみます。

▼ targetTripIDs に含まれている trip_id の route_id を求める

```
targetRouteIDs=[]
def tripFilter(inGtfs,inTrip):
```

```
if inTrip.trip_id in targetTripIDs:  
    if inTrip.route_id not in targetRouteIDs:  
        targetRouteIDs.append(inTrip.route_id)  
    return False  
gtfs.trips.filter(tripFilter, update=False)
```

tripFilter 関数の構成は先程示した stopTimesFilter 関数と同様です。filter メソッドの呼び出しについても、update=False と指定しているので、この処理においても gtfs 変数に格納されている GTFS-JP の情報は何ら変化しません。

6.2.4 対象の route_id に限定した routes 情報を作り保存する

ここまで来ればあとは簡単です。routes 情報から targetRouteIDs に含まれている route_id のみを残し、それ以外は削除してしまえば十分です。言い換えれば、残すべきか否かを確認する対象の route_id が targetRouteIDs に含まれていれば True を返し、それ以外は False を返す述語を filter に与えることに他なりません。この述語は以下のように記述できます。

▼targetRouteIDs に含まれているか否かを調べる述語

```
lambda _, route: route.route_id in targetRouteIDs
```

今回のこの述語では gtfs の情報は使いませんので、最初の引数は _ (アンダーバー) としています (とはいえ、アンダーバーでなくとも Python では特に警告なども出ませんので、適当な引数名を与えておけば十分です)。

また、今回は gtfs 変数に格納されている GTFS-JP の情報を更新しますので filter メソッドの update 引数には True を設定しなければなりませんが、この引数にはデフォルト値として True が設定されていますので、明示的に指定する必要はありません。

既に先の節で例として示していますが、実際の呼び出しが以下のようになります：

```
gtfs.routes.filter(lambda _, route: route.route_id in targetRouteIDs)
```

保存については例えば

```
gtfs.save('filteredGtfs.zip')
```

とすれば filteredGtfs.zip として gtfs 変数に格納されている GTFS-JP の情報は保存されます。

以上をまとめた全体のプログラムリストは次のようになります。このプログラムでは対象の GTFS-JP ファイルとして秋田中央交通が提供しているファイルを用い、「美術」という文字列を含むバス停を対象のバス停として指定しています。出力する GTFS-JP ファイルのファイル名には"bidaiRelatedGTFS.zip" を指定しています。入力ファイル名と出力ファイル名はプログラムリスト冒頭にある srcGtfsFilePath ならびに dstGtfsFilePath にて指定していますので、適宜差し替えて使って下さい。また、対象のバス停の stop_id については targetBusStopIDs に格納していますので、こちらも必要に応じて差し替えて使ってみて下さい。

▼ 対象の stop_id を通過する経路のみを取り出す

```
import egGTFS

srcGtfsFilePath='bus-akitachuoukotsu.zip'
dstGtfsFilePath='bidaiRelatedGTFS.zip'

gtfs=egGTFS.open(srcGtfsFilePath)

targetBusStopIDs=['6043_01','6043_02']

targetTripIDs=[]
def stopTimesFilter(inGtfs,inStopTime):
    if inStopTime.stop_id in targetBusStopIDs:
        if inStopTime.trip_id not in targetTripIDs:
            targetTripIDs.append(inStopTime.trip_id)
    return False
gtfs.stop_times.filter(stopTimesFilter,update=False)

targetRouteIDs=[]
def tripFilter(inGtfs,inTrip):
    if inTrip.trip_id in targetTripIDs:
```

```
if inTrip.route_id not in targetRouteIDs:  
    targetRouteIDs.append(inTrip.route_id)  
return False  
gtfs.trips.filter(tripFilter, update=False)  
assert len(targetRouteIDs)>0  
  
gtfs.routes.filter(lambda _, route: route.route_id in targetRouteIDs)  
gtfs.save(dstGtfsFilePath)
```

【コラム】フィルタ処理のテスト

作成したプログラムが正しく動作していることを確かめる方法には様々なものがあります。上で示したプログラムにより、正しく情報がフィルタされているのか否かを調べるために地図上に GTFS-JP に含まれている全ての経路を描画してみましょう。

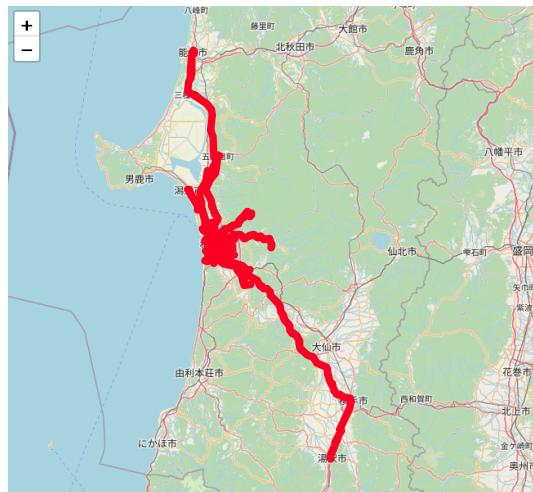
サンプルプログラムとして同梱されている ex_drawAllRoute.py を用いれば

```
python ex_drawAllRoute.py 対象のGTFS-JPファイル 出力するHTMLファイル
```

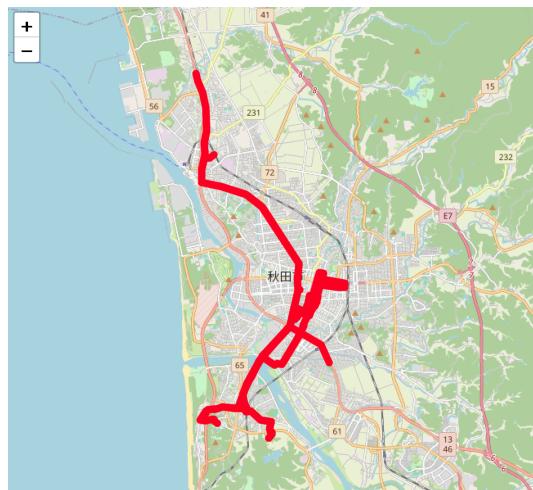
とすることにより、指定した GTFS-JP に含まれている全ての経路を Web ブラウザで確認できるようになります。

フィルタする前の（つまり元となった）GTFS-JP ファイルと、フィルタ後の GTFS-JP ファイルを指定しそれぞれ HTML ファイルを出力させてブラウザ上に表示すると次の図のようになります。

ex_drawAllRoute.py では描画した経路を考慮し地図の縮尺を調整しますので、それぞれの画像の縮尺にも注意してみて下さい。フィルタ後は秋田市を中心とする地図となっていますが、フィルタ前のものでは秋田市ののみならず隣接する市なども描画されていることが確認できます。このことからも、今回作成したプログラムにて経路がフィルタされていることが確認できると思います。



▲図 6.1: フィルタ前



▲図 6.2: フィルタ後

第7章

egGTFS の活用例

ここまで、egGTFS の基本的な使い方や地図との連携について説明してきました。この章では、より実践的なデータ処理へ egGTFS を活用する方法を示してみたいと思います。

7.1 路線バスの利用可能範囲の概算

とある自治体を走る路線バスは、その自治体のどれくらいの範囲を利用可能範囲（カバレッジ）として持つのでしょうか？ ここでは問題を単純化するために、バス停の半径 r メートル以下の範囲がそのバス停の利用可能範囲として計算してみます。

プログラムで計算する場合、次のように処理してゆけば良さそうです。

1. GTFS-JP 内に存在する全てのバス停のうち、対象の自治体に設置されているものののみをフィルタ
2. フィルタ後のバス停の位置を中心とし、半径 r の円を各バス停毎に生成
3. 上記で生成された円群から構成される図形の面積を計算

7.1.1 バス停の住所を求める

GTFS-JP に含まれている stops.txt の各行には緯度と経度の情報が含まれています。この情報から住所が得られれば、そのバス停がどの自治体に存在しているのかが分かります。このような処理は逆ジオコーディングなどとも呼ばれており、egGTFS でも利用している GeoPy モジュールにて提供されています。なお、GeoPy は egGTFS のインストール時に必要に応じてインストールされます。そのため、ここで改めてインストールの作業は必要ありません。

緯度経度から市の名前を求める方法

練習のため、まずはGeoPyを使って特定の緯度経度からその地点の住所を求めてみましょう。ジオコーディングにはGeoPyに含まれているNominatimのインスタンスを用いることとします。例として、ここでは秋田駅の住所を調べてみましょう。

秋田駅の緯度経度はGoogle Mapによるとおおよそ北緯39.7165 東経140.1281とのことです。次のプログラムでは、reverseメソッドにこの座標を与え、市の名前を求めています。

▼緯度経度から市の名前を求める

```
from geopy.geocoders import Nominatim  
geoLocator=Nominatim(user_agent="egGTFS_example")  
location=geoLocator.reverse("39.7165,140.1281") # 緯度経度  
print(f"市の名称は「{location.raw['address']['city']}」です。")
```

実行結果は

```
市の名称は「秋田市」です。
```

となり、無事正しい情報が得られています。

なお、

```
print(location.address)
```

とすると、location変数に格納されている情報を確認できます。

```
秋田停車場線, 千秋久保田町, 渋坂, 秋田市, 秋田県, 010-8530, 日本
```

必要に応じて活用すると良いでしょう。

7.1.2 stop_id から市の名前を取得する

GeoPy の逆ジオコーディング機能を用いて、GTFS-JP に含まれているバス停のみを集めてみます。次に示すプログラムでは targetGtfsFilePath に対する GTFS-JP ファイルへのパスを指定し、targetCityName に対する市名を設定すれば、指定した市内に存在するバス停のみを targetBusStopID に格納します。

例として、秋田中央交通における秋田市内のバス停の数ならびに秋田市外に存在するバス停の数を表示するようにしています（市外にあるバス停の stop_id は outsideBusStopIDs に格納するようにしています）。

なお、'city' というキーの存在を確かめているのは、郡における町などは 'town' というキーに町名が格納されており、その場合は 'city' というキーが存在しないためです。

正しくプログラムが動作していることを確認するため、市内・市外にあると判断したバス停からランダムに stop_id を選び、それらの緯度経度とバス停の名前を表示するようにもしています。

また、GeoPy を用いた逆ジオコーディングは処理時間がかかるため、バス停をひとつ処理する際にドットをひとつ出力するようにしています（環境によっては 10 分程度かかる場合もあります）。

▼ 市内と市外にあるバス停を分類する

```
from geopy.geocoders import Nominatim
geoLocator=Nominatim(user_agent="egGTFS_example")

import egGTFS

srcGtfsFilePath='bus-akitachuoukotsu.zip'
targetCityName='秋田市'

gtfs=egGTFS.open(srcGtfsFilePath)
targetBusStopIDs=[]
outsideBusStopIDs=[]

def addressFilter(_,inStop):
    print('.',end='',flush=True)
    t=geoLocator.reverse(f'{inStop.stop_lat},{inStop.stop_lon}')
```

```
if 'city' not in t.raw['address'] \
    or t.raw['address']['city']!=targetCityName:
    outsideBusStopIDs.append(inStop.stop_id)
else:
    targetBusStopIDs.append(inStop.stop_id)
return False

gtfs.stops.filter(addressFilter,update=False)
print()
print(f"{targetCityName} 内にあるバス停の数={len(targetBusStopIDs)}")
print(f"{targetCityName} 外にあるバス停の数={len(outsideBusStopIDs)}")

# 念のため確認する
def printStopInfo(inStopID):
    stopInfo=gtfs.stops[inStopID]
    print(f"バス停の名称 : {stopInfo.stop_name}")
    print("バス停の住所 : ")
    location=geoLocator.reverse(f"{stopInfo.stop_lat},{stopInfo.stop_lon}")
    print(location)

import random
insideStopID=random.choice(targetBusStopIDs)
outsideStopID=random.choice(outsideBusStopIDs)

print()
print(f"{targetCityName} 内にあるバス停の例 :")
printStopInfo(insideStopID)

print()
print(f"{targetCityName} 外にあるバス停の例 :")
printStopInfo(outsideStopID)
```

実行結果は次のとおりです：

```
.....
( 中略 )
.....
秋田市 内にあるバス停の数=1082
秋田市 外にあるバス停の数=92
```

```
秋田市 内にあるバス停の例 :
```

バス停の名称：高陽青柳町

バス停の住所：

山王商店街, 八橋本町, 秋田市, 秋田県, 010-0967, 日本

秋田市 外にあるバス停の例：

バス停の名称：自動車学校前

バス停の住所：

潟上市, 秋田県, 010-0124, 日本

出力結果を見てみると正常に動作しているようです。

7.1.3 バス停を中心とする円形領域の和の面積を求める

対象の自治体に存在するバス停のみを取り出すことができましたので、次はこれらのバス停を中心とする円形領域の和からなる領域の面積を求めてみたいと思います。

shapely の紹介と使い方

それぞれの円が重なり合わない状況であれば、各円の面積に対象のバス停の個数を乗ずれば対象領域の面積の計算は終了します。

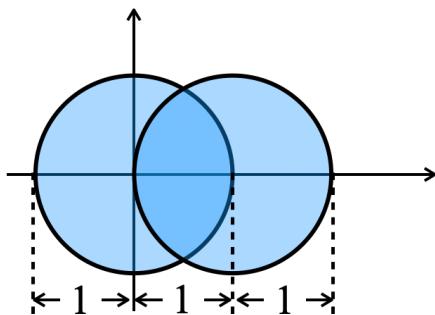
しかし今回は半径を様々な値に変更することを考えていますので、円の領域が重なる場合も想定して計算プログラムを作ってみます。このような領域の面積計算はそれなりに手間がかかるのですが、幸いにも shapely というモジュールを使うと簡単に近似値を計算できます。

shapely のインストールは

```
pip install shapely
```

などとして下さい。

無事にインストールできたら、簡単な計算をさせてみましょう。次のプログラムでは、円の中心が 1だけ離れた半径 1の重なり合う 2つの円の面積を計算しています。もちろん、重なり合う部分は 2重に計算はしません（なので、計算結果は 2つの円の面積 2π から重なっているレンズ状の領域部分の面積を減じた値となります）。



▲図 7.1: テストプログラムで計算する円領域

```
from shapely.geometry import Point
from shapely.ops import unary_union
import math

circles=[{"center":(0,0),"radius":1}, {"center":(1,0),"radius":1}]
]

unionCircle=[Point(c['center']).buffer(c['radius']) for c in circles]
area=unary_union(unionCircle).area

print("shapely による近似値=",area)
print("手計算による理論値 =",4*math.pi/3+math.sqrt(3)/2)
```

理論値は $\frac{4\pi}{3} + \frac{\sqrt{3}}{2}$ ですので、それも合わせて表示するようにしています。
実行結果は次のとおりです。

```
shapely による近似値= 5.0481120537579285
手計算による理論値 = 5.054815608570829
```

少々誤差がありますが、ここではこの方法で得られた値を近似値として用います。

路線バスの利用可能範囲面積を計算するプログラム

いよいよ GeoPy による逆ジオコーディングと shapely による領域計算を組み合わせて、路線バスの利用可能範囲面積を計算するプログラムを作ってみましょう。基本的にはこれまで説明した技術をつなぎ合わせただけです。

先程のプログラムと少し変更している点は、targetBusStopIDs に追加する際に、別途そのバス停の緯度と経度を targetPos に記録している点です。

地理に関する情報処理では座標系が重要となります、今回は近似計算を行うとし、座標系は緯度経度の空間（度形式 DD）をそのまま用いることとします（少々乱暴ですが…）。

バス停を中心とする円の半径についてですが、緯度 1 度に関する距離は約 111 km と知られていますので、こちらも少々乱暴ですが、今回はこの値を利用することとします。よって、半径 R メートルは DD 形式上では $R/111000$ の半径として処理を行います。同様に、1 平方度は 111^2 キロ平方メートルですので、shapely で得られた値にこの値を乗じてキロ平方メートルに変換しています。

以上を踏まえ、プログラムは次のようになります。なお、各バス停の利用可能範囲は変数 R に格納している数値（単位はメートル）を用いて計算します。

【コラム】度形式と DMS 形式

今回のプログラムでは緯度経度を度形式（DD = Decimal Degrees 形式）を用いています。度形式というのは度を単位としたもので、0.50 度に 0.20 度を足したら 0.70 度になるというものです。

そんなの当たり前なのでは？と思われるかもしれません、緯度経度の表し方については度分秒で表す DMS 形式（Degree Minute Second 形式）もあります。度の位は 10 進数なのですが、度以下の値については時計と同様の 60 進法となります。なので、 $0^\circ 50'00''$ （0 度 50 分 0 秒）に $0^\circ 20'00''$ を加えたら $1^\circ 10'00''$ となります。

このように DMS 形式で表された数値を計算機で扱う場合には少し注意が必要となります。

▼利用可能範囲の計算プログラム

```
from geopy.geocoders import Nominatim
geoLocator=Nominatim(user_agent="egGTFS_example")

import egGTFS

srcGtfsFilePath='bus-akitachuoukotsu.zip'
targetCityName='秋田市'
R=400 # 単位はメートルで指定

gtfs=egGTFS.open(srcGtfsFilePath)
targetBusStopIDs=[]
targetPos=[]

outsideBusStopIDs=[]

def addressFilter(_,inStop):
    print('.',end='',flush=True)
    lat=inStop.stop_lat
    lon=inStop.stop_lon
    t=geoLocator.reverse(f"{lat},{lon}")
    if 'city' not in t.raw['address'] \
        or t.raw['address']['city']!=targetCityName:
        outsideBusStopIDs.append(inStop.stop_id)
    else:
        targetBusStopIDs.append(inStop.stop_id)
        targetPos.append((lat,lon))
    return False

gtfs.stops.filter(addressFilter,update=False)
print()
print(f"{targetCityName} 内にあるバス停の数={len(targetBusStopIDs)}")
print(f"{targetCityName} 外にあるバス停の数={len(outsideBusStopIDs)}")

# バス停を中心とした円領域の和領域を作成
from shapely.geometry import Point
from shapely.ops import unary_union

radius=float(R)/111000
unionCircle=[Point(pos).buffer(radius) for pos in targetPos]
area=unary_union(unionCircle).area
print(f"利用可能範囲 {area*111*111} km^2")
```

このプログラムを実行すると、最終的には次のような表示が出ます。

利用可能範囲 115.13334282407028 km²

今回は秋田県秋田市を対象に秋田中央交通さんの路線について、各バス停から400 メートルの範囲をそれぞれのバス停の利用可能範囲としました。これは成人の歩く速度がおよそ 1 分あたり 80 メートルとされていますので、5 分程度で到着できる範囲をそれぞれのバス停の利用可能範囲としています（ちなみに 2014 年に国土交通省都市局都市計画課が発表した「都市構造の評価に関するハンドブック」ではバス停を中心とする半径 300 メートルを公共交通沿線地域としています）。

秋田市の公式サイトによると秋田市の面積は約 906 キロ平方メートルだそうです。計算してみると約 13 % 弱となり、（街路などの形状を一切無視した乱暴な議論となり恐縮ですが）少なくとも 8 割強の場所においては、5 分以上歩かなければバス停に到着できないことが分かります。

このプログラムは、あくまでも近似値を計算するものですので、さらなる精度が必要な場合は使用できません。しかし、他の地域との比較や、新しい経路の導入を検討する場合などにおいては、活用できるのではないかと思っています。

【コラム】処理時間の高速化

バス停からの距離を変化させて計算する場合、上で示したプログラムでは都度住所を調べるため、処理に時間がかかることがあります。

当該の自治体に存在する stop_id が収集できれば十分ですので、対象のバス停を収集し、その情報をファイルに書き出すプログラムと、そのファイルから stop_id の情報を取り出し、計算するプログラムを分離するなどの工夫が考えられます。

このようにすれば、計算毎に（時間のかかる）バス停の住所を取り出す処理が省略できますので、各種の条件における計算処理の大幅な時間短縮を図ることができます。

7.2 経路長の計算

空間的アクセシビリティ指標のひとつに路線長密度というものが存在します。これは、バスや鉄道の路線長を可住地面積で割った値です。このような指標を計算するためには各路線における全体の路線長の値が必要となります。

GTFS-JP の仕様においては、路線を取り扱うために `jp_parent_route_id` が設けられていますが、ここでは単純に、指定された `route_id` を持つ経路の長さを求めてみます。ここでは、`shape` 情報を使わずに関係するバス停の直線距離を用いて近似的に計算する方法と、`shape` 情報を用いて、より実際の走行距離に近い値を計算する 2 つのプログラムを作成してみます。

それぞれのプログラムは、次のように作れば良さそうです。

shape 情報を使わない場合

1. 指定された経路 (`route_id`) 情報と紐づく便 (`trip_id`) を探す
2. 見つけた `trip_id` から停車するバス停の停車順リストを作る
3. 作成した停車順リストを用いてバス停間の直線距離を算出しつつその総和を求める

shape 情報を使う場合

1. 指定された経路 (`route_id`) 情報と紐づく便 (`trip_id`) を探す
2. 見つけた `trip_id` から `shape_id` を求める
3. 得られた `shape_id` から、その `shape` の全長を求める

では、それぞれのプログラムを実際に作ってみましょう。

7.2.1 shape 情報を使わずに経路長を求めるプログラム

関係する `trip_id` を求める

指定された `route_id` を持つ便情報の検索は `trips.txt` 内の情報において、各行の `route_id` を調べれば十分です。

今回は指定した route_id を持つ全ての便情報を収集する必要はありませんので filter 関数は使いません。単純に逐次確認を行い、見つかったらその処理を中断するといった方法にて調査します。

汎用的に使えるようにするために、調査する route_id は targetRouteID 変数にて指定することとします。

```
import egGTFS
gtfs=egGTFS.open('bus-akitachuoukotsu.zip')

targetRouteID='新屋線_A'
for trip in gtfs.trips:
    if trip.route_id==targetRouteID:
        targetTripID=trip.trip_id
        break
print('targetTripID=',targetTripID)
```

このプログラムを実行すると、次のような表示が出力されました。

```
targetTripID= 平日_06時40分_系統71002
```

問題なさそうですので、この trip_id を用いてバス停間の距離を用いて路線長の近似値を求めてみたいと思います。

停車するバス停の情報を収集する

trip_id が分かったので、次は stop_times.txt の情報を用いて停車するバス停の情報を順次取得します。幸い egGTFS では既に説明しているように、stop_times は trip_id をキーとして検索できますし、それにより得られた stop_times の情報は stop_sequence が昇順になるよう並び替えられた配列となります。そのためプログラムは随分と単純なものとなります。

```
for t in gtfs.stop_times[targetTripID]:
    print(t.stop_sequence,gtfs.stops[t.stop_id].stop_name)
```

追加したこのプログラムは、tagetTripID に格納されている trip_id に関して停車する順番でバス停の情報を表示するものです。実際に実行してみると次のような出力が得られます。

```
1 西部サービスセンター  
2 新屋駅前  
3 新屋駅入口  
  :  
( 中略 )  
  :  
26 中通二丁目  
27 買物広場  
28 秋田駅西口
```

停車するバス停の緯度経度より距離を求める

あとはこれらのバス停の位置を用いて、バス停間の距離を計算し、その総和をもって路線長の近似値とすれば十分です。

バス停の位置は stops.txt の該当情報の stop_lat および stop_lon に格納されています。緯度経度で表示された 2 点間の距離は Geopy の geodesic を用いて計算できます。

```
from geopy.distance import geodesic

firstBusStopID=gtfs.stop_times[targetTripID][0].stop_id
startBusStop=gtfs.stops[firstBusStopID]
prePos=(startBusStop.stop_lat,startBusStop.stop_lon)
total=0.0
for t in gtfs.stop_times[targetTripID]:
    stop=gtfs.stops[t.stop_id]
    pos=(stop.stop_lat,stop.stop_lon)
    distance=geodesic(prePos,pos)
    total+=distance.km
    prePos=pos
print(f"total length={total} km")
```

実際に実行してみると次のような結果が得られます。

```
total length=7.342485133156174 km
```

全てをまとめたソースコードは以下のようになります：

```

import egGTFS
gtfs=egGTFS.open('bus-akitachuoukotsu.zip')

targetRouteID='新屋線_A'
for trip in gtfs.trips:
    if trip.route_id==targetRouteID:
        targetTripID=trip.trip_id
        break
print('targetTripID=',targetTripID)

for t in gtfs.stop_times[targetTripID]:
    print(t.stop_sequence,gtfs.stops[t.stop_id].stop_name)

from geopy.distance import geodesic

firstBusStopID=gtfs.stop_times[targetTripID][0].stop_id
startBusStop=gtfs.stops[firstBusStopID]
prePos=(startBusStop.stop_lat,startBusStop.stop_lon)
total=0.0
for t in gtfs.stop_times[targetTripID]:
    stop=gtfs.stops[t.stop_id]
    pos=(stop.stop_lat,stop.stop_lon)
    distance=geodesic(prePos,pos)
    total+=distance.km
    prePos=pos
print(f"total length={total} km")

```

7.2.2 shape 情報を用いて経路長を求めるプログラム

次はより精度の高い shapes.txt の情報を用いて経路長を求めてみましょう。shapes.txt は GTFS-JP ファイルに必ず含まれるわけではありませんが、同梱されている場合は経路長計算の精度を高めるのに役立ちます。

対象の trip_id を求める方法は shape 情報を使わない場合と同様ですので割愛します。

対象の shpe_id を求める

trip_id が判明したら、その trip_id を持つ trips.txt の行には shape_id に関する情報も含まれていますので、それを求めれば十分です。

```
targetShapeID=gtfs.trips[targetTripID].shape_id  
print('targetShapeID=',targetShapeID)
```

実際に実行してみると次のような表示が得られ、無事 shape_id が取得できている様子が確認できます。

```
targetShapeID= 71002
```

shape_id から経路長を求める

egGTFS では shape_id を指定して shape 情報を取得すると、指定した shape_id に対応する情報が shape_pt_sequence の値に関して昇順で整列された配列として返されます。

また shapes.txt の各行は緯度経度を有していますので、バス停間の距離を求めたのと同様の処理を行い、経路長を求められます。

```
from geopy.distance import geodesic  
  
firstPoint=gtfs.shapes[targetShapeID][0]  
prePos=(firstPoint.shape_pt_lat,firstPoint.shape_pt_lon)  
total=0.0  
for shape in gtfs.shapes[targetShapeID]:  
    pos=(shape.shape_pt_lat,shape.shape_pt_lon)  
    distance=geodesic(prePos,pos)  
    total+=distance.km  
    prePos=pos  
print(f"total length={total} km")
```

実際の出力は次のようになります：

```
total length=8.23608420654469 km
```

先程の shape 情報を用いない方法では、バス停間を直線距離で結んだものの総和を経路長として算出していました。もちろん、実際には直線で移動はできませんので、より実際の経路に近い shape 情報を用いた距離の方が長くなるはずです。今回の結果でも、shape 情報を用いた値の方がより長い経路長となっています。

こちらも全体のプログラムを以下に示しておきます。

```
gtfs=egGTFS.open('bus-akitachuoukotsu.zip')

targetRouteID='新屋線_A'
for trip in gtfs.trips:
    if trip.route_id==targetRouteID:
        targetTripID=trip.trip_id
        break
print('targetTripID=',targetTripID)

targetShapeID=gtfs.trips[targetTripID].shape_id
print('targetShapeID=',targetShapeID)

from geopy.distance import geodesic

firstPoint=gtfs.shapes[targetShapeID][0]
prePos=(firstPoint.shape_pt_lat,firstPoint.shape_pt_lon)
total=0.0
for shape in gtfs.shapes[targetShapeID]:
    pos=(shape.shape_pt_lat,shape.shape_pt_lon)
    distance=geodesic(prePos,pos)
    total+=distance.km
    prePos=pos
print(f"total length={total} km")
```

7.3 指定したバス停における待ち時間の分析

最後に指定したバス停における待ち時間について分析するプログラムを書いてみましょう。バス停が置かれている場所により、どのような違いがあるのかが分かると簡単なサービスギャップ分析も可能となります。

ここでは、指定した stop_id を持つバス停に対し、便毎の最小待ち時間と最大待ち時間を計算してみます。

プログラムの概略は次のようにになります。

1. 指定した stop_id を持つ情報を stop_times から収集
2. 得られた情報を trip_id 毎に整理
3. 各 trip_id に対し、最大待ち時間と最小待ち時間を計算

それでは早速作成していきましょう。

7.3.1 stop_id から stop_times の情報を集める

対象のバス停にどのような便がいつやってくるのかを調べるために、stop_times.txt の情報を集めます。なお、調査対象のバス停の stop_id は、targetStopID 変数に格納されているものとします。

バス停にバスがやってくる時間は、経路ならびに平日・土休日の組により識別します。そのため、route_id と service_id をキーとする連想配列（辞書）を用意し、それぞれのキーに対し該当する到着時刻を保存するようにします。以下のプログラムでは timesByRoute 変数でこの辞書を取り扱っています。

```
import egGTFS
gtfs=egGTFS.open('bus-akitachuoukotsu.zip')

targetStopID='6043_01'
print('targetBusStop name=',gtfs.stops[targetStopID].stop_name)

timesByRoute=dict()
def genDict(gtfs,stopTime):
    if stopTime.stop_id==targetStopID:
        tripID=stopTime.trip_id
        routeID=gtfs.trips[tripID].route_id
        serviceID=gtfs.trips[tripID].service_id
        if (routeID,serviceID) not in timesByRoute:
            timesByRoute[(routeID,serviceID)]=[]
        timesByRoute[(routeID,serviceID)].append(stopTime.arriv
    return False
gtfs.stop_times.filter(genDict,update=False)

for k,v in timesByRoute.items():
    for arrivalTime in v:
        print(k,arrivalTime)
```

上記のプログラムを実行すると次のような表示が得られました。

```
('新港線_A', '平日') 07:45:00
('新屋線_A', '平日') 07:28:00
('新屋線_A', '平日') 08:18:00
:
(中略)
:
('新屋線_A', '年末・年始') 16:48:00
('新屋線_A', '年末・年始') 17:48:00
('新屋線_A', '年末・年始') 19:48:00
```

データが少ないですが、複数の経路からなるバスの時刻が得られていることが分かります（最初の行は新「港」線_Aであり、2行目以降は新「屋」線_Aです）。

7.3.2 停車時刻からバスがやってくる頻度を求める

それぞれの経路毎のバスの到着時刻は、特に到着順序を意識して集めたわけではありません。バスの到着頻度を計算する前に、これらの時刻を昇順に整列しておきます。幸い、arrival_time は 24 時間表示の文字列になっていますので、リストに対し sort メソッドを呼び出せば十分です。

```
for k,v in timesByRoute.items():
    v.sort()
    for arrivalTime in v:
        print(k,arrivalTime)
```

実行してみると、正しく時刻が昇順に整列していることが分かります。

次にこの整列された時刻から、到着時刻の間隔の最大値と最小値を求めてみます。時刻の演算については、egGTFS に存在する Time クラスが利用できますのでそれを使用します。

なお、経路によっては 1 日に 1 回しかバスがやってこない場合もあります。その場合は、最大値と最小値を 24 時間として設定することとします。

時間間隔を計算する部分は次のようなプログラムとなります：

```
for k,v in timesByRoute.items():
    if len(v)<2:
        print(f"経路:{k} 最大=最小=24 時間")
        continue
    minTimeDiff=egGTFS.TimeDiff(24,0,0)
    maxTimeDiff=egGTFS.TimeDiff(0,0,0)
    v.sort()
    t1=egGTFS.Time(v[0])
    for arrivalTime in v:
        t2=egGTFS.Time(arrivalTime)
        if t1==t2: continue
        dt=t2-t1
        if minTimeDiff>dt: minTimeDiff=dt
        if maxTimeDiff<dt: maxTimeDiff=dt
        t1=t2
    print(f"経路:{k}")
    print(f" 最大時間間隔={maxTimeDiff}")
    print(f" 最小時間間隔={minTimeDiff}")
    print()
```

実行してみると次のような表示が得られます：

```
経路:('新港線_A', '平日') 最大=最小=24 時間
経路:('新屋線_A', '平日')
    最大時間間隔=02:01:00
    最小時間間隔=00:14:00

経路:('新屋線_A', '平日\u3000大森山あり')
    最大時間間隔=01:25:00
    最小時間間隔=00:40:00

    :
    (中略)
    :

経路:('新屋線_A', '年末・年始')
    最大時間間隔=02:00:00
    最小時間間隔=00:30:00
```

実際に調べてみると随分待ち時間の間隔に違いがあることが分かります。

これらの情報を収集する場合、もちろん手作業で行うことも可能です。しかし、手作業での情報収集には時間がかかり誤りも発生します。一方、プログラムを使った処理にはプログラムの設計誤りという別のリスクが伴います。しかし適切にテストを施せば、信頼性の高い結果を得ることができます。egGTFSは厳密な検証を行ったソフトウェア部品ではありません。それでも本書で示した程度の品質は有しています。egGTFSは一見複雑に思える公共交通データの解析や改善作業を、より効率的で正確なものに変えることができるのではないかと思っています。

この章では egGTFS を活用した様々な処理を紹介しました。また、GeoPy や folium などのモジュールも紹介し、その使い方の一例を具体的に示しました。

拙作の egGTFS が公共交通の分析や改善などにお役に立てそうであれば、作者としては嬉しく思います。

付録 A

メタプログラミングと egGTFS

egGTFS の重要な機能として GTFS-JP 内に含まれる情報を同ファイル内に存在する同名のオブジェクトとして取り扱えるようにすることができます。そのため、例えば routes クラスでは routes[route_id] という使い方で routes.txt に含まれる当該 route_id を持つ行の情報を取得できるように設計されています。egGTFS では、この routes.txt の各行の値を保持するクラスを routes_record として定義しています。

routes_record クラスでは、routes.txt の各行で保存されている route_id や route_type 等の各列に対する情報を該当の列と同名のプロパティ（フィールド）の値として提供しなければなりません。これは今、例として挙げた routes クラスのみならず、stops や agency 等 GTFS-JP の仕様書にて定義されている CSV ファイル全てに対して求められることとなります。

GTFS-JP 内に格納されている個々の CSV ファイルに対応するように実装してしまうとプログラムコードの量も増え、見通しも悪くなってしまいます（その結果、保守性も悪くなってしまいます）。そのため egGTFS ではメタプログラミングを活用し、列名を指定するだけでそれぞれのプロパティをプログラムにより動的に実装するようにしています。

これらの仕組みを担っているのは SingleRecord ならびに RecordSet というクラスです。これらはそれぞれ単一の行（データ）を保持する CSV ファイル、複数行を保持する CSV ファイルのために用意されている抽象クラスです。egGTFS ではこれらのクラスを用意し、活用することで必要なコード量を減らしています。

以下に示すのは egGTFS にて実際に使われている routes クラスの定義です。

```
class routes(RecordSet):
    def __init__(self,inZipFileObj):
        super().__init__(inZipFileObj,'routes.txt',
                         ['route_id','agency_id',
                          'route_short_name','route_long_name',
                          'route_desc','route_type','route_url','route_col'
                         >or',
                          'route_text_color','jp_parent_route_id'],
                          'route_id',routes_record)

class routes_record(Record): pass
```

たったこれだけで routes クラスならびに個々の route 情報を表す routes_record クラスの定義は終わりです。

では GTFS-JP の仕様書で定義されている個々の情報の取扱を担っている SingleRecord ならびに RecordSet というクラスはどのように定義されているのでしょうか？

これら 2 つのクラスではいずれも（個々の）行が含む列名のリストを引数として受け取り、ヘルパークラスである indexSet クラスのコンストラクタに渡しています。このクラスは route_id などのフィールド名をインデックス値（0,1,2 … などの整数値）に変換するためのクラスです。

A.1 単一行の場合 (SingleRecord を用いる場合)

SingleRecord から派生しているクラス（例えば agency クラス等）では、 setattr 関数を用いて当該のクラスの属性値を動的に生成しています（実際にこれらの処理は addGettersForSingle 関数にて行われています）。

A.2 複数行の場合（RecordSet を用いる場合）

一方、routes のように複数の行を保持する CSV ファイルを取り扱うクラスでは、クラス単位で setattr 関数を用いる方法は採用していません。

そのため、SingleRecord を用いる場合とは異なり、inRecordClass で指定されるクラスのインスタンスにそれぞれの列名でそれぞれの値が取得できるようになる必要があります。

RecordSet のコンストラクタでは

```
if inRecordClass!=None:  
    inRecordClass.fieldNameList=inFieldNameList  
    inRecordClass.index=self.index  
def recordInit(self,inArray):  
    self.record=inArray  
    for fieldName in self.fieldNameList:  
        colNo=getattr(self.index,fieldName)  
        self.__dict__[fieldName]=self.record[colNo] if colNo>=0 else None  
inRecordClass.__init__=recordInit  
inRecordClass.__str__=lambda self: str(self.record)
```

という処理を行い、引数として渡された各行を表すクラス（例えば routes_record クラス等）のコンストラクタとして recordInit 関数を設定しています。

コンストラクタとして実行される recordInit 関数では、`__dict__` を用いて routes_record 等の各行を表すクラスに設定し、これを用いて route_id などのプロパティ（フィールド）を動的に設定するようにしています。

このように egGTFS では全体の記述力を減らすために動的なプロパティ生成を用いています。構造としてはそれほど複雑なものではないので、必要に応じてカスタマイズして利用してみて下さい。

付録 B

API リファレンス

この付録では egGTFS に実装されている関数やクラスのメソッドのうち主なものを紹介しています。Pythonにおいては、インスタンスに作用するメソッドは第 1 引数に `self` 等インスタンス自身を示す変数を記す必要がありますが、本リファレンスでは実際の利用に即した記述をしているため、それらは省略しています。

B.1 egGTFS モジュールの関数

B.1.1 egGTFS.open(gtfsFilePath)

引数の `gtfsFilePath` には GTFS-JP ファイル (ZIP ファイル形式) へのパスを指定します。戻り値は `egGTFS` クラスのインスタンスです。

B.2 egGTFS クラス

本モジュールの中心的な役割を担うクラスです。コンストラクタを明示的に使うのではなく、上で説明しているファクトリ関数 (`egGTFS.open` 関数) を使用してインスタンスを得て下さい。

B.2.1 クラスマソッド

コンストラクタ egGTFS(inGtfsZipFilePath)

`inGtfsZipFilePath` には GTFS-JP ファイル (ZIP ファイル形式) へのパスを指定します。戻り値は `egGTFS` クラスのインスタンスです。

B.2.2 インスタンスマソッド

save(inOutputZipFilePath)

指定されたパスに現在の GTFS-JP の情報を書き出します。

version()

egGTFS のバージョンを文字列にて返します。

getAllStopsMap()

全てのバス停が記載された地図 (folium の Map インスタンス) を返します。

getShapeMap(inShapeID,weight=8,color="#FF0000")

inShapeID に指定された走行ルートを描画した地図 (folium の Map インスタンス) を返します。描画する直線の太さは引数の weight にて、色は color にて指定できます。

getTripMap(inTripID,weight=8,color="#0000FF")

指定された便情報 (inTripID) に関する走行ルートを描画した地図 (folium の Map インスタンス) を返します。こちらも描画する線の太さは weight にて、色は color にて指定できます。

drawShape(inMap,inShapeID,weight=8,color="#FF0000")

inMap に指定された地図 (folium の Map インスタンス) に、inShapeID で指定された走行ルートを描画します。このメソッドも描画する線の太さは weight にて、色は color にて指定できます。

getBusPos(inTripID,inHour_or_TimeStr,inMinute=None, inSecond=None,epsilon=0.00003):

inTripID にて指定された便情報の指定された時刻 (inHour_or_TimeStr および inMinute, inSecond) のバスの位置を推測して返します。epsilon はバスの位置を計算する際の許容誤差で、デフォルトでは約 3m を指定しています。

getShapeIdByTripID(inTripID)

指定された便情報 (trip ID) に関する shape ID を返します。

makeName(inName)

地図上に、バス停等の名前を表示する際に利用するユーティリティメソッドです。inName には文字列を指定します。渡された文字列を以下の span タグで囲った文字列を返します：～

B.3 Time クラス

時刻を表すクラスです。

B.3.1 クラスマソッド

コンストラクタ Time

Time("hh:mm:ss") 形式もしくは Time(hour,minute,second) 形式にてインスタンスを生成できます。

B.3.2 インスタンスマソッド

文字列化

str 関数や print 関数など、必要に応じた文字列化処理にも対応しています。

演算

等号、時刻の大小比較に対応しています。加減算ならびに乗算に対応しています。加算は後述する TimeDelta 型との演算が実装されており、減算は TimeDelta 型ならびに Time 型に対応しています。

Time 型と TimeDelta 型との和は Time 型となります。Time 型と TimeDelta 型との減算結果は Time 型に、Time 型同士の減算結果は TimeDiff 型となります。

乗算に関しては Time 型と数値との計算に対応しており、乗算結果は Time 型となります。

B.4 TimeDelta クラス

Time クラスの派生クラスです。TimeDelta クラスに特有のメソッドはありません。

B.5 TimeDiff クラス

Time クラスの派生クラスです。TimeDiff クラスに特有のメソッドはありません。

B.6 AreaRect クラス

矩形領域を表すクラスです。folium で地図を表示する際に利用できます。

B.6.1 クラスマソッド

コンストラクタ

デフォルトコンストラクタのみが用意されています。インスタンスを生成する場合は `a=AreaRect()` 等とし、`minLat` や `minLat,maxLat,minLon,maxLon` というプロパティ（フィールドもしくはメンバ変数）に値を代入して下さい（例：`a.maxLat=0` 等）。それぞれ最大・最小の緯度ならびに経度を示すことを意図しています。

B.6.2 インスタンスマソッド

`union(inAreaRect)`

引数で指定された AreaRect インスタンスを含む矩形領域を表すよう、矩形領域を拡張します。

applyScale(inScale)

矩形領域の中心を維持したまま、指定された倍率だけ矩形領域を拡大または縮小します（inScale が 1 未満の場合は縮小となります）。

getBounds()

`[[self.minLat, self.minLon], [self.maxLat, self.maxLon]]` なるリストを返します。

B.7 GTFS-JP 内の情報を表すクラス

GTFS-JP は複数の CSV ファイルをまとめた ZIP 形式のファイルです。以下に示すクラスは、これら CSV ファイルの情報を取り扱うクラスです。特に記述がない場合は、積極的に使用をおすすめするコンストラクタやインスタンスメソッドを有していないことを意味します。

複数行からなる CSV ファイルの場合、各行の情報を表すクラスは当該の CSV ファイルのファイル名に `_record` を付したものになります。例えば `stops` であれば `stops_record` となります。

B.7.1 agency, agency_jp, stops, routes, routes_jp, trips, office_jp, calendar, calendar_dates, fare_attributes, fare_rules, frequencies, transfers, feed_info, translations クラス

GTFS-JP で定義されている列名をメンバ変数として有するクラスです。

B.7.2 stop_times クラス

`stop_times[trip_id]` として、`trip_id` を指定することにより関連する `stop_times` の情報が配列形式で取得できます。この配列には `trip_id` に関連する `stop_times` の各行の情報がまとめられています。この情報は、配列の添字の 0 番から順次 `stop_sequence` の値が昇順になるように並び替えられていますので、ユーザー側で再度並べ替える必要はありません。

B.7.3 shapes クラス

`shapes[shape_id]` として、`shape_id` を指定することにより関連する `shapes` の情報を配列形式で取得できます。`stop_times` と同様に、この配列には `shape_id` に関する `shapes` の各行の情報がまとめられています。この情報は、配列の添字の 0 番から順次 `shape_pt_sequence` の値が昇順になるように並び替えられていますので、ユーザー側で再度並べ替える必要はありません。

あとがき

egGTFS のはじめての入門書となる本書はいかがだったでしょうか。今回はできるだけ読みやすい文章を書くため、推敲作業に ChatGPT を活用してみました。その効果はありましたでしょうか？ 少しでも理解しやすい文章になつていれば幸いです。

作者の立場から言うと、ChatGPT の文章は私の好みとは異なっており、そのため AI が出力した文章をそのまま使う状況とはなりませんでした。それでも作者としては推敲の良いきっかけを与えてくれる便利な道具であったと感じています。つくづく 21 世紀に生きているんだなあ、と実感しています。

閑話休題。egGTFS は、できるだけ使いやすく、かつ、コンパクトなソフトウェアライブラリを目指して作成しています。この小さなプログラムが皆様のお役に立ちそうでしょうか。もし、有用に感じて頂けたのであれば作者としては大変嬉しく思います。ソフトウェアにはバグや不具合がつきものです。egGTFS に関するバグについてはもとより、GTFS-JP に格納されている情報の解釈や取り扱いについて、ならびに各種の指標に関しての誤解や間違い等があれば [iigura @ akibi.ac.jp](mailto:iigura@akibi.ac.jp) までご連絡いただければ幸いです。

路線バスを取り巻く環境は地域ごとに様々な差異があるとは思いますが、egGTFS を利用することにより、皆さんの地域において路線バスが更に便利になることを願っています。

著者紹介

秋田公立美術大学教授。人工神経回路網の理論解析から並列処理むけのスクリプト言語開発まで、ソフトウェア開発と数学を用いて各種の研究を行っています。平成 28 年度日本リモートセンシング学会 学会賞（論文賞）や、若い頃はソニーミュージックエンタテインメントの Digital Entertainment Program'96 等、いくつかの受賞歴を有しています。博士（情報学）です。

表紙の画像は本書のために新たに作成したジェネラティブアート作品を素材として用いています。

egGTFS 入門ガイド

2024 年 3 月 29 日 ver 1.0 (GTFS 活用ソフトウェアの開発に関する研究)

著 者 飯倉宏治

© 2024 飯倉宏治