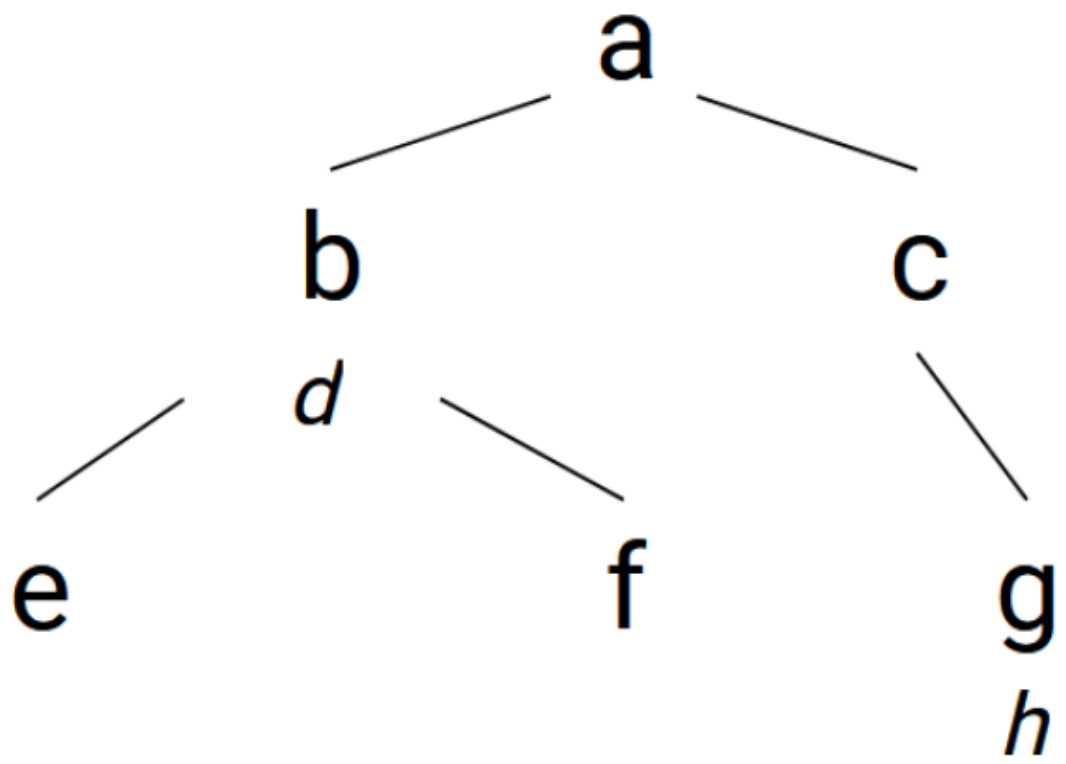
Artículo
Nancy Martínez · 16 abr, 2020 Lectura de 10 min

API nativa de IRIS para Python

A partir de la versión 2019.2, InterSystems IRIS ofrece su API nativa para Python como un método de alto rendimiento para acceso a datos. La API nativa permite interactuar directamente con la estructura de datos nativa de IRIS.

Globals

Como desarrolladores de InterSystems, seguramente ya estais familiarizados con los globals. Vamos a revisar los aspectos básicos por si os apetece un repaso, pero podéis saltar a la siguiente sección si no lo necesitáis.InterSystems IRIS usa globals para almacenar los datos. Un global es una matriz dispersa formada por nodos que pueden, o no, tener un valor y subnodos. Lo que sigue es un ejemplo abstracto de un global:



En este ejemplo, a es un nodo raíz, al que nos referimos con el *nombre del global*. Cada nodo tiene una *dirección de nodo,* que consiste en el nombre del global y uno o múltiples *subscripts* (nombres de subnodos). a tiene los subscripts b y c; la dirección de nodo de esos nodos es s a->b y a->c.

Los nodos a->b y a->c->g tienen un valor (d y h), los nodos a->b->e y a->b->f no tienen valor. El nodo a->b tiene subscripts e y f.

Podéis encontrar una descripción detallada de esta estructura en el libro de InterSystems "Using Globals".

Leer y escribir en el Global

La API nativa de Python permite leer y escribir los datos directamente en el global de IRIS. El paquete irisnative

está disponible en GitHub - o si tienes InterSystems IRIS instalado de forma local en tu equipo, lo encontrarás en el subdirectorio dev/python de tu directorio de instalación.

La función irisnative.createConnection te permite crear una conexión con IRIS y la función irisnative.createIris te ofrece un objeto de esta conexión con el que se puede manipular el global. Este objeto tiene un método get and set para leer/escribir desde/hacia el global, y un método kill para eliminar un nodo y sus subnodos. También tiene un método isDefined, que devuelve 0 si el nodo solicitado no existe; 1 si tiene un valor, pero ningún descendiente; 10 si no tiene valor y tiene descendientes; u 11 si tiene un valor y descendientes.

import irisnative conn = irisnative.createConnection("127.0.0.1", 51773, "USER", "", " ") iris = irisnative.createIris(conn) iris.set("value", "root", "sub1", "sub2") # sets "value" to root->sub1->sub2 print(iris.get("root", "sub1", "sub2")) print(iris.isDefined("root", "sub1")) iris.kill("root") conn.close()

También tiene un método iterator para hacer un bucle sobre los subnodos de un determinado nodo. (Mostraré su uso en la siguiente sección)

Podéis leer una descripción completa de cada método en la documentación de la API nativa para Python.

Archivos de datos de tránsito del GTFS de San Francisco

Almacenamiento de datos en un global

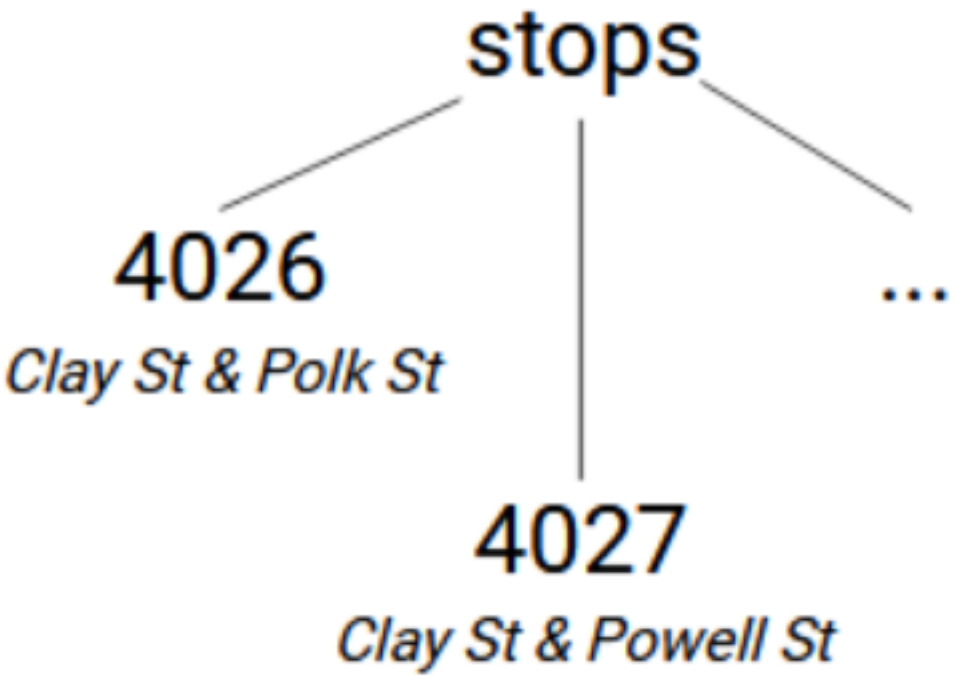
El General Transit Feed Specification (GTFS) es un formato para horarios y rutas de transporte público. Vamos a ver cómo podemos usar la API nativa de IRIS para trabajar con datos del GTFS de San Francisco a partir del 10 de junio de 2019.

Primero, almacenaremos en el global IRIS la información de los archivos de datos. (En esta demostración no usaremos todos los archivos y columnas). Los archivos están en formato CSV, en el que la primera fila muestra los nombres de las columnas, y las otras filas contienen los datos. En Python, comenzaremos con las importaciones necesarias y estableceremos una conexión con IRIS:

import csv import irisnative conn = irisnative.createConnection("127.0.0.1", 51773, "USER", "", " ") iris = irisnative.createIris(conn)

En base a los nombres de las columnas y los datos, podemos construir una estructura de árbol razonable para cada archivo y usar iris.set para almacenar los datos en el global.

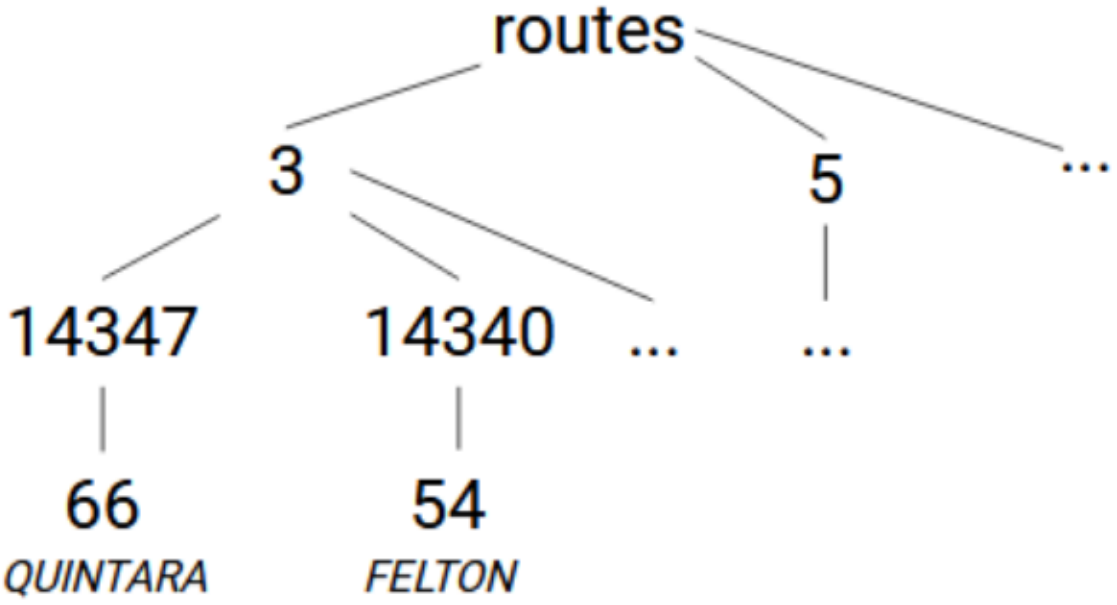
Empecemos con el archivo stops.txt, que contiene todas las paradas (stops) del transporte público en la ciudad. De este archvo, solo usaremos las columnas stopid y stopname. Las almacenaremos en un global llamado stops dentro de una estructura de árbol con una capa de nodos, con los IDs de parada como subscripts y los nombres de paradas como valores de nodos. Así, nuestra estructura tendrá este aspecto stops → [stopid]=[stopname]. (Para este artículo, usaré los corchetes para indicar cuándo un subscript no es literal, sino que es un valor leído de los archivos de datos).



with open("stops.txt", "r") as csvfile: reader = csv.reader(csvfile) next(reader) # Ignore column names # stops -> [stopid]=[stopname] for row in reader: iris.set(row[6], "stops", row[4])

csv.reader devuelve un iterador de listas que contienen los valores separados por comas. La primera línea contiene los nombres de las columnas, por lo que la omitiremos con
next(reader). Usaremos iris.set para definir el nombre de la parada como valor de stops -> [stopid].

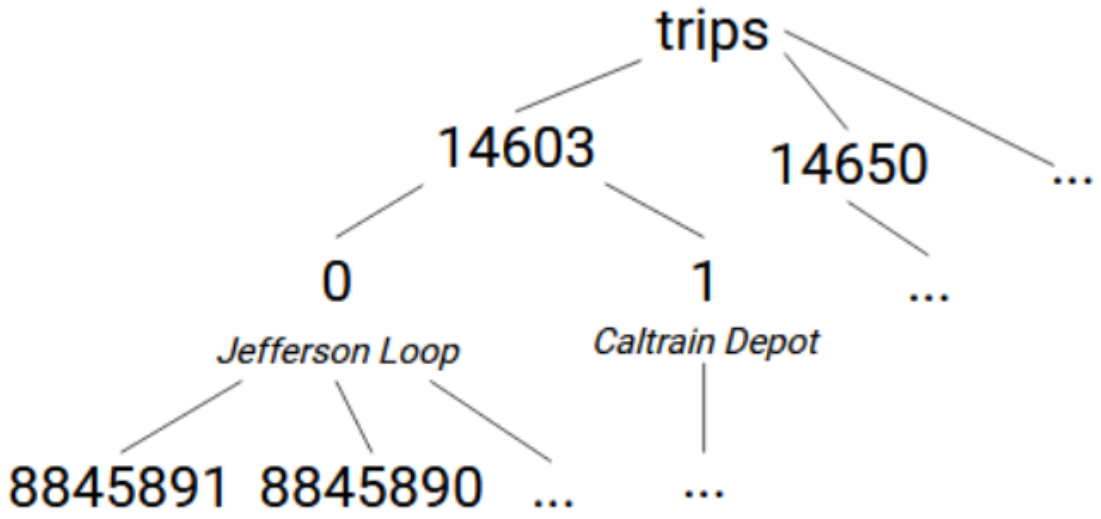
A continuación viene el archivo routes.txt, del que usaremos las columnas routetype, routeid, routeshortname y routelongname. Una estructura de global razonable es routes -> [routetype] -> [routeid] -> [routeshortname]=[routelongname]. (El tipo de ruta (route) es 0 para un tranvía, 3 para un autobús y 5 para un tranvía). Podemos leer el archivo CSV y colocar los datos en el global exactamente de la misma forma.



with open("routes.txt", "r") as csvfile: reader = csv.reader(csvfile) next(reader) # Ignore column names # routes -> [routetype] -> [routeid] -> [routeshortname]=[routelongname] for row in reader: iris.set(row[0], "routes", row[1], row[5], row[8])

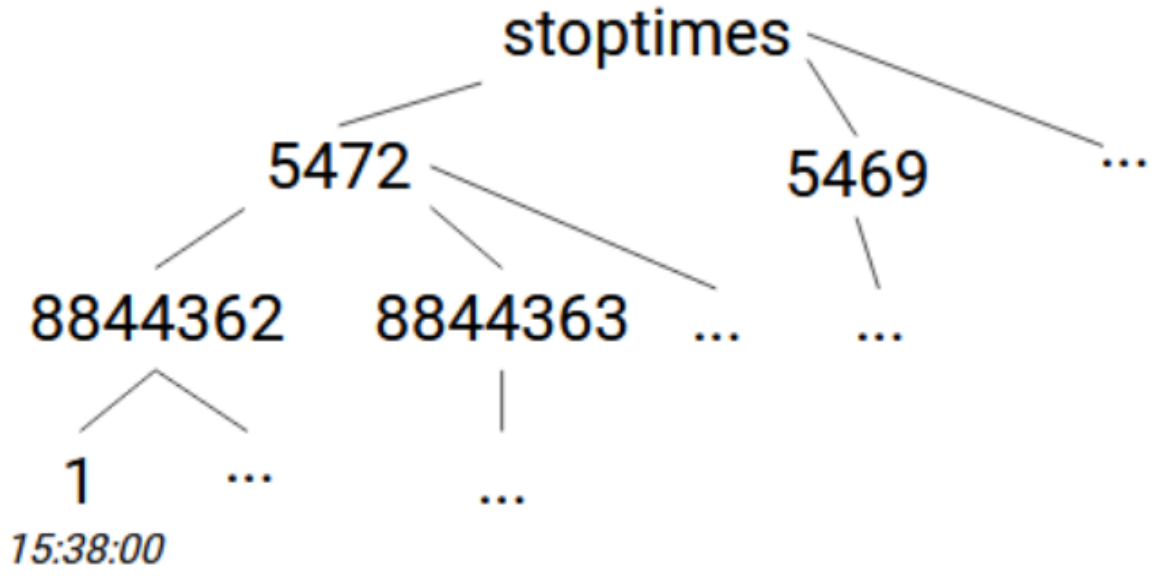
Cada ruta tiene *viajes* (trips), almacenados en trips.txt, del que usaremos las columnas routeid, directionid, tripheadsign and tripid. Los viajes se identifican de forma única con su ID de viaje ("trip ID"), que veremos más adelante en el archivo de horarios de parada. Los viajes de una ruta pueden separarse en dos grupos según su dirección, y las direcciones tienen letreros ("head signs") asociados a ellas. Esto nos lleva a la estructura de árbol trips -> [routeid] -> [directionid]=[tripheadsign] -> [tripid].

Aquí necesitamos dos llamadas a iris.set - una para definir el valor al nodo ID de dirección, y otra para crear el nodo sin valor del ID del viaje.




```
with open("trips.txt", "r") as csvfile: reader = csv.reader(csvfile) next(reader) # Ignore column names # trips -> [routeid] -> [directionid]=[tripheadsign] ->[tripid] for row in reader: iris.set(row[3], "trips", row[1], row[2]) iris.set(None, "trips", row[1], row[2], row[6])
```

Por último, leeremos y almacenaremos los horarios de las paradas. Se almacenan en stoptimes.txt y usaremos las columnas stopid, tripid, stopsequence y departuretime. Una primera opción podría ser usar stoptimes -> [stopid] -> [tripid] -> [departuretime] o, si queremos mantener la secuencia de paradas, stoptimes -> [stopid] -> [tripid] -> [stopsequence]=[departuretime].




```
with open("stoptimes.txt", "r") as csvfile: reader = csv.reader(csvfile) next(reader) # Ignore column names # stoptimes -> [stopid] -> [tripid] -> [stopsequence]=[departuretime] for row in reader: iris.set(row[2], "stoptimes", row[3], row[0], row[4])
```

Consultar los datos usando la API nativa

A continuación, nuestro objetivo es **encontrar todos los horarios de salida para la parada con el nombre dado.**

Primero, recuperamos el ID de parada desde el nombre de parada dado, y luego usaremos ese ID para encontrar los horarios relevantes en stoptimes.

La llamada iris.iterator("stops") nos permite iterar sobre los subnodos del nodo raíz de paradas. Queremos iterar sobre los pares de subscripts y valores (para comparar los valores con el nombre dado, e inmediatamente conocer el subscript si coinciden), por lo que llamamos a .items() en el iterador, lo que define la clase de devolución como tuplas (de subscripts, valores). Luego podemos iterar sobre todas estas tuplas y encontrar la parada correcta.

stopname = "Silver Ave & Holyoke St" iter = iris.iterator("stops").items() stopid = None for item in iter: if item[1] == stopname: stopid = item[0] break if stopid is None: print("Stop not found.") import sys sys.exit()

Merece la pena observar que buscar una clave por su valor mediante iteración no es muy eficiente si hay muchos nodos. Una forma de evitar esto sería tener otra matriz, en la que los subscripts sean los nombres de las paradas y los valores sean sus ID. El valor --> búsqueda clave consistiría entonces en una consulta a esta nueva matriz.

De forma alternativa, puede usar el nombre de parada como identificador en cualquier parte del código en vez del ID de parada - el nombre de parada también es único.

Como puedes ver, si tenemos una cantidad de paradas significativa, esta búsqueda puede tardar bastante. También se le conoce como "full scan" (análisis completo). Pero podemos aprovecharnos de los globals y construir la matriz invertida, en la que los nombres serán claves con IDs como valores.

iter = iris.iterator("stops").items() stopid = None for item in iter: iris.set(item[0], "stopnames", item[1])

Tener el global de nombres de paradas (stopnames), en el que el índice es el nombre y el valor es el ID, cambiará el código anterior para encontrar el stopid por nombre. El código será el siguiente, que se ejecutará sin una búsqueda de análisis completo:

stopname = "Silver Ave & Holyoke St" stopid=iris.get("stopnames", stopname) if stopid is None: print("Stop not found.") import sys sys.exit()

En este punto, podemos encontrar los horarios de parada. El subárbol stoptimes -> [stopid] tiene los ID de viajes como subnodos, que tienen los horarios de paradas como subnodos. No nos interesan los ID de los viajes, solo los horarios de parada, por lo que iteraremos sobre todos los ID de viajes y recopilaremos todos los horarios de paradas para cada uno de ellos.

allstoptimes = set() trips = iris.iterator("stoptimes", stopid).subscripts() for trip in trips: allstoptimes.update(iris.iterator("stoptimes", stopid, trip).values())

Aquí no estamos usando .items() en el iterador, pero usaremos .subscripts() y .values() porque los IDs de viajes son subscripts (sin valores asociados)([stopsequence]=[departuretime]), solo nos interesan los valores y horas de salida. La llamada .update agrega todos los elementos del iterador a nuestro conjunto existente. El nuevo conjunto ahora contiene todos los horarios (únicos) de paradas:

for stoptime in sorted(allstoptimes): print(stoptime)

Ahora vamos a hacerlo un poco más complicado. En vez de encontrar *todas* las horas de salida para una parada, encontraremos únicamente las horas de salida para una parada *de una ruta dada* (en ambas direcciones), cuyo ID de ruta es dado. El código para encontrar el ID de parada a partir del nombre de parada puede preservarse por completo. Entonces, se recuperarán todos los ID de paradas de la ruta dada. Estos ID se usan entonces como una restricción adicional al recuperar las horas de salida.

El subárbol de trips -> [routeid] se parte en dos direcciones, que tienen todo los ID de viajes como subnodos.

Podemos iterar sobre las direcciones como antes, y agregar todos los subnodos de las direcciones a un conjunto.

route = "14334" selectedtrips = set() directions = iris.iterator("trips", route).subscripts() for direction in directions: selectedtrips.update(iris.iterator("trips", route, direction).subscripts())

En el siguiente paso, queremos encontar los valores de todos los subnodos de stoptimes -> [stopid] -> [tripid] donde [stopid] es el ID de parada recuperado y [tripid] es cualquiera de los IDs de viaje seleccionados. Iteramos sobre el conjunto selectedtrips para encontrar todos los valores relevantes:

allstoptimes = set() for trip in selectedtrips: allstoptimes.update(iris.iterator("stoptimes", stopid, trip).values()) for stoptime in sorted(allstoptimes): print(stoptime)

Un último ejemplo muestra el uso de la función isDefined. Ampliaremos el código escrito anteriormente: en lugar de realizar un "hard-code" del ID de ruta, se ofrece el nombre corto de una ruta y luego el ID de la ruta se deberá recuperar en base a esto. Los nodos con los nombres de rutas están en la capa inferior del árbol. La capa de arriba contiene los ID de rutas. Podemos iterar sobre todos los tipos de rutas, luego sobre todos los ID de rutas y, si el nodo routes -> [routetype] -> [routeid] -> [routeshortname] existe y tiene un valor (isDefined returns 1), sabemos entonces que [routeid] es el ID que buscamos.

routeshortname = "44" route = None types = iris.iterator("routes").subscripts() for type in types: routeids = iris.iterator("routes", type).subscripts() for routeid in routeids: if iris.isDefined("routes", type, routeid, routeshortname) == 1: route = routeid if route is None: print("No route found.") import sys sys.exit()

El código sirve como sustituto de la línea route = "14334" en "hard-code".

Una vez finalizadas todas las operaciones de IRIS, podemos cerrar la conexión a la base de datos:

conn.close()

Próximos pasos

Hemos explicado cómo se puede usar la API nativa para Python para acceder a la estructura de datos de InterSystems IRIS. Después, lo hemos aplicado a los datos de transporte público de San Francisco. Para profundizar más en la API, podéis visitar esta documentación. La API nativa también está disponible para Java, .NET y Node.js.

#API #Python #InterSystems IRIS

URL de fuente:https://es.community.intersystems.com/post/api-nativa-de-iris-para-python