Article
Thomas Daniels · Sep 3, 2019 9m read

IRIS Native API for Python

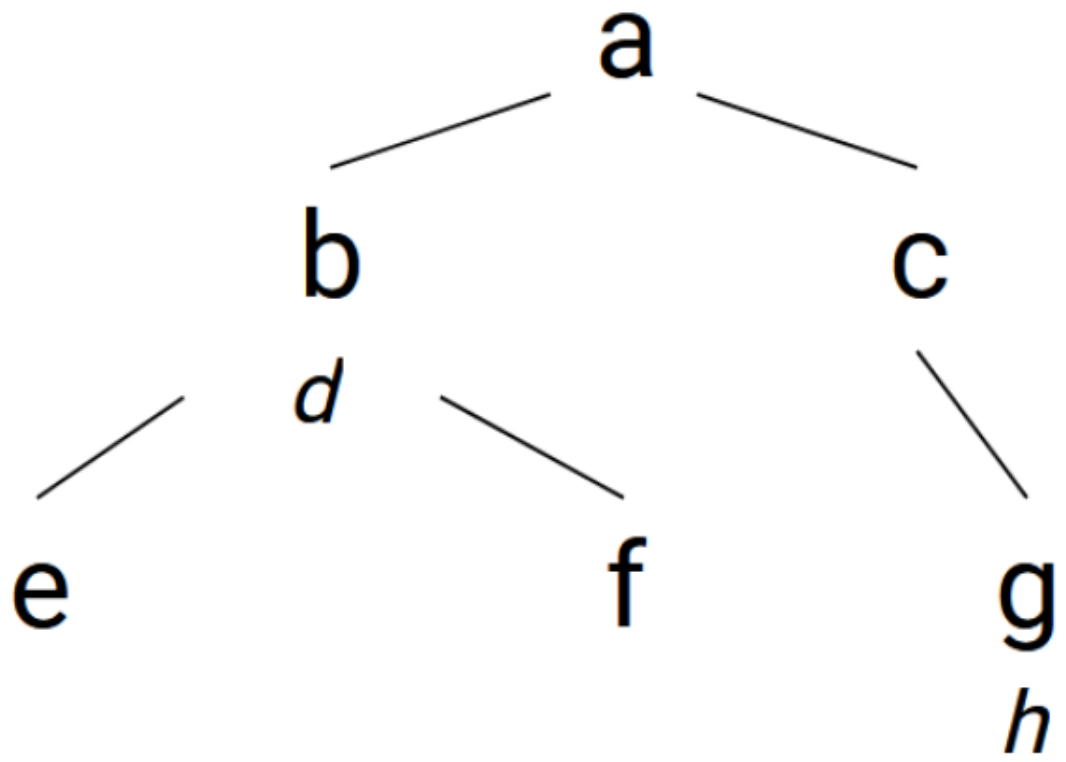
Introduction

Since version 2019.2, InterSystems IRIS has provided their Native API for Python as a high-performance data access method. The Native API allows you to directly interact with the native IRIS data structure.

Globals

As InterSystems developers, you ' re likely already familiar with the globals. We ' ll review the basics in case you ' d like a refresher, but feel free to skip ahead to the next section.

InterSystems IRIS uses globals to store the data. A global is a sparse array which consists of nodes that may or may not have a value and subnodes. The following is an abstract example of a global:



In this example, a is a root node, referred to as the *global name*. Every node has a *node address* that consists of the global name and one or multiple *subscripts* (names of subnodes). a has subscripts b and c; the node address of those nodes is a->b and a->c.

The nodes a->b and a->c->g have a value (d and h), the nodes a->b->e and a->b->f are valueless. The node a->b has subscripts e and f.

An in-depth description of this structure can be found in the InterSystems book "Using Globals".

Reading and Writing to the Global

The Native Python API allows direct reading and writing of data to the IRIS global. The irisnative package is available on GitHub — or if InterSystems IRIS is locally installed on your machine, you ' ll find it in the dev/python subdirectory of your installation directory.

The irisnative.createConnection function lets you create a connection to IRIS and the irisnative.createIris function gives you an object from this connection with which we can manipulate the global. This object has a get and set method to read/write from/to the global, and a kill method to delete a node and its subnodes. It also has an isDefined method, which returns 0 if the requested node does not exist; 1 if it has a value, but no descendants; 10 if it is valueless and has descendants; or 11 if it has a value and descendants.

```
import irisnative
conn = irisnative.createConnection("127.0.0.1", 51773, "USER", "<user>", "<password>"
)
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()
```

It also has an iterator method to loop over subnodes of a certain node. (Usage will be demonstrated in the next section.)

For a full description of each method, refer to the API documentation.

San Francisco GTFS Transit Data Files

Storing the data in global

The General Transit Feed Specification (GTFS) is a format for public transportation schedules and routes. Let ' s see how we can use the IRIS Native API to work with San Francisco GTFS data from June 10, 2019.

First, we will store the information from the data files in IRIS global. (Not all files and columns will be used in this demo.) The files are in CSV format, where the first row shows the column names and all other rows contain the data. In Python, we will start with the necessary imports and establishing a connection to IRIS:

```
import csv
import irisnative

conn = irisnative.createConnection("127.0.0.1", 51773, "USER", "<user>", "<password>"
)
iris = irisnative.createIris(conn)
```

Based on the column names and data, we can construct a sensible tree structure for each file and use iris.set to store the data in the global.

Let ' s start with the stops.txt file, which contains all public transport stops in the city. From this file, we will only use the stopid and stopname columns. We will store them in a global named stops within a tree structure with one layer of nodes, with the stop IDs as subscripts and the stop name as node values. So our structure looks like stops → [stopid]=[stopname]. (For this article, I ' ll use square brackets to denote when a subscript is not literal, but instead a value read from the data files.)

```
with open("stops.txt", "r") as csvfile:
```

```
    reader = csv.reader(csvfile)
    next(reader)  # Ignore column names

    # stops -> [stop_id]=[stop_name]
    for row in reader:
        iris.set(row[6], "stops", row[4])
```

csv.reader returns an iterator of lists that hold the comma-separated values. The first line contains the column names, so we will skip it with next(reader). We will use iris.set to set the stop name as value of stops -> [stopid].

Next is the routes.txt file, of which we will use the routetype, routeid, routeshortname and routelongname columns. A sensible global structure is routes -> [routetype] -> [routeid] -> [routeshortname]=[routelongname]. (The route type is 0 for a tram, 3 for a bus, and 5 for a cable car.) We can read the CSV file and put the data in the global in exactly the same way.

```
with open("routes.txt", "r") as csvfile:
    reader = csv.reader(csvfile)
    next(reader)  # Ignore column names

    # routes -> [route_type] -> [route_id] -> [route_short_name]=[route_long_name]
    for row in reader:
        iris.set(row[0], "routes", row[1], row[5], row[8])
```

Every route has *trips*, stored in trips.txt, of which we will use the routeid, directionid, tripheadsign and tripid columns. Trips are uniquely identified by their trip ID (which we will later see in the stop times file). Trips on one route can be separated into two groups based on their direction, and the directions have head signs associated with them. This leads to the tree structure trips -> [routeid] -> [directionid]=[tripheadsign] -> [tripid].

We need two iris.set calls here — one to set the value to the direction ID node, and one to create the valueless node of the trip ID.

```
with open("trips.txt", "r") as csvfile:
    reader = csv.reader(csvfile)
    next(reader)  # Ignore column names

    # trips -> [route_id] -> [direction_id]=[trip_headsign] ->[trip_id]
    for row in reader:
        iris.set(row[3], "trips", row[1], row[2])
        iris.set(None, "trips", row[1], row[2], row[6])
```

Lastly, we will read and store the stop times. They ' re stored in stoptimes.txt and we will use the stopid, tripid, stopsequence and departuretime columns. A first option could involve using stoptimes -> [stopid] -> [tripid] -> [departuretime] or if we want to keep the stop sequence, stoptimes -> [stopid] -> [tripid] -> [stopsequence]=[departuretime].

```
with open("stop_times.txt", "r") as csvfile:
    reader = csv.reader(csvfile)
    next(reader)  # Ignore column names

    # stoptimes -> [stop_id] -> [trip_id] -> [stop_sequence]=[departure_time]
    for row in reader:
        iris.set(row[2], "stoptimes", row[3], row[0], row[4])
```

Querying the Data Using the Native API

Next, our goal is to **find all departure times for the stop with the given name**.

First, we retrieve the stop ID from the given stop name, then we will use that ID to find the relevant times in the stoptimes.

The iris.iterator("stops") call lets us iterate over the subnodes of the stops root node. We want to iterate over the pairs of subscripts and values (to compare the values with the given name, and immediately know the subscript if they match), so we call .items() on the iterator, which sets the return type to (subscript, value) tuples. We can then iterate over all these tuples and find the right stop.

```
stop_name = "Silver Ave & Holyoke St"

iter = iris.iterator("stops").items()

stop_id = None

for item in iter:
    if item[1] == stop_name:
        stop_id = item[0]
        break

if stop_id is None:
    print("Stop not found.")
    import sys
    sys.exit()
```

It is worth noting that looking up a key by its value through iteration is not very efficient if there are a lot of nodes. One way to avoid this would be to have another array, where the subscripts are the stop names and the values are the IDs. The value --> key lookup would then consist of one query to this new array.

Alternatively, you could use the stop name as identifier everywhere in your code instead of the stop ID -- the stop name is unique as well.

As you can see, if we have a significant amount of stops, this search can take a while — it is also known as " full scan " . But we can take advantage of globals and build the inverted array where names will be keys with IDs for values.

```
iter = iris.iterator("stops").items()

stop_id = None

for item in iter:
    iris.set(item[0], "stopnames", item[1])
```

Having the global of stopnames, where index is name and value is ID, will change the code above to find the stopid by name to the following code which will run without a full scan search:

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

At this point, we can find the stop times. The subtree stoptimes -> [stopid] has trip IDs as subnodes, which have the stop times as subnodes. We are not interested in the trip IDs — only the stop times — so we will iterate over all trip IDs and collect all stop times for each of them.

```
all_stop_times = set()

trips = iris.iterator("stoptimes", stop_id).subscripts()
for trip in trips:
    all_stop_times.update(iris.iterator("stoptimes", stop_id, trip).values())
```

We are not using .items() on the iterator here, but we will use .subscripts() and .values() because the trip IDs are subscripts (without associated values) or the bottom layer ([stopsequence]=[departuretime]), we are only interested in the values and departure times. The .update call adds all items from the iterator to our existing set. The set now contains all (unique) stop times:

```
for stop_time in sorted(all_stop_times):
    print(stop_time)
```

Let ' s make it a little more complicated. Instead of finding *all* departure times for a stop, we will find only departure times for a stop *for a given route* (both directions) where the route ID is given. The code to find the stop ID from the stop name can be kept in its entirety. Then, all trip IDs on the given route will be retrieved. These IDs are then used as an extra restriction when retrieving departure times.

The subtree of trips -> [routeid] is split in two directions, which have all trip IDs as subnodes. We can iterate over the directions as before, and add all of the directions ' subnodes to a set.

```
route = "14334"

selected_trips = set()

directions = iris.iterator("trips", route).subscripts()
for direction in directions:
    selected_trips.update(iris.iterator("trips", route, direction).subscripts())
```

As a next step, we want to find the values of all subnodes of stoptimes -> [stopid] -> [tripid] where [stopid] is the retrieved stop ID and [tripid] is any of the selected trip IDs. We iterate over the selectedtrips set to find all relevant values:

```
all_stop_times = set()

for trip in selected_trips:
    all_stop_times.update(iris.iterator("stoptimes", stop_id, trip).values())

for stop_time in sorted(all_stop_times):
    print(stop_time)
```

A final example shows the usage of the isDefined function. We will expand on the previously written code: instead of hardcoding the route ID, the short name of a route is given, then the route ID has to be retrieved based on that. The nodes with the route names are on the bottom layer of the tree. The layer above contains the route IDs. We can iterate over all route types, then over all route IDs, and if the node routes -> [routetype] -> [routeid] -> [routeshortname] exists and has a value (isDefined returns 1), then we know that [routeid] is the ID we ' re looking for.

```
route_short_name = "44"
route = None

types = iris.iterator("routes").subscripts()
```

```
for type in types:
    route_ids = iris.iterator("routes", type).subscripts()
    for route_id in route_ids:
        if iris.isDefined("routes", type, route_id, route_short_name) == 1:
            route = route_id

if route is None:
    print("No route found.")
    import sys
    sys.exit()
```

The code serves as a replacement for the hardcoded route = "14334" line.

When all IRIS operations are done, we can close the connection to the database:

```
conn.close()
```

Next Steps

We ’ ve covered how the native API for Python can be used to access the data structure of InterSystems IRIS, then applied to San Francisco public transport data. For a deeper dive into the API, you can visit the documentation. The native API is also available for Java, .NET and Node.js.

#API #Python #InterSystems IRIS