

Article

[Guillaume Rongier](#) · Jul 22, 2022 10m de lecture

## IRIS native API pour Python

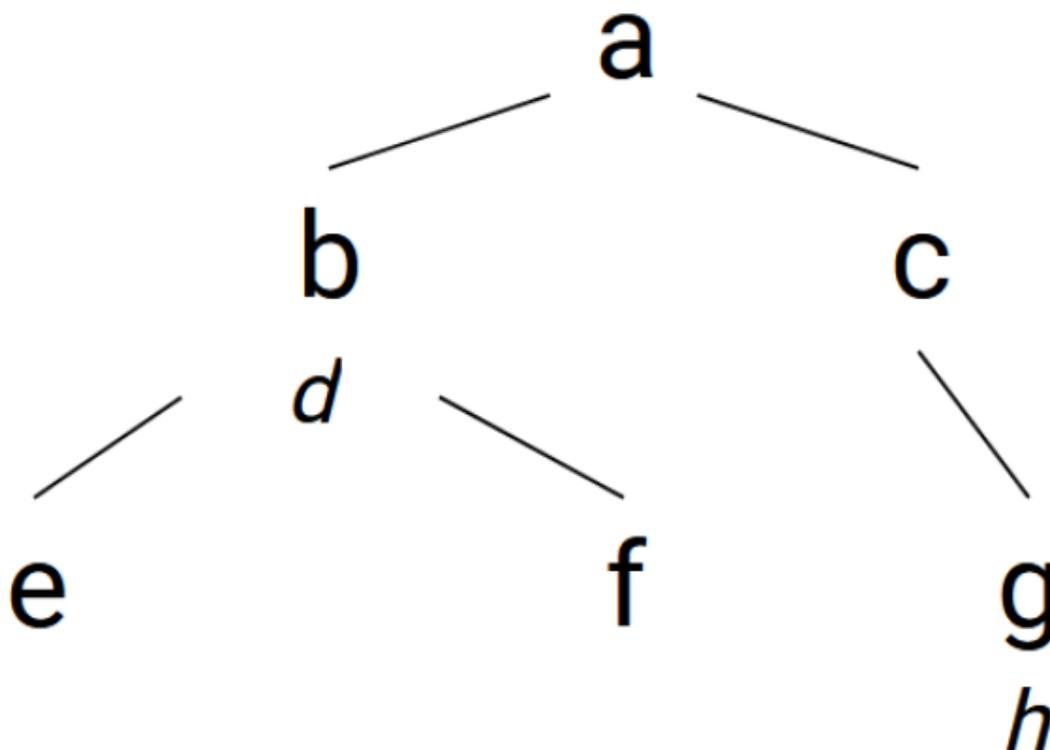
### Introduction

Depuis la version 2019.2, InterSystems IRIS fournit son API native pour Python comme méthode d'accès aux données haute performance. L'API native vous permet d'interagir directement avec la structure de données IRIS native.

### Globales

En tant que développeurs InterSystems, vous êtes probablement déjà familiarisés avec les globales. Nous allons passer en revue les bases au cas où vous souhaiteriez un rafraîchissement, mais n'hésitez pas à passer directement à la section suivante.

InterSystems IRIS utilise des globales pour stocker les données. Une globale est un tableau clairsemé qui se compose de nœuds qui peuvent ou non avoir une valeur et de sous-nœuds. Ce qui suit est un exemple abstrait d'une globale :



Dans cet exemple, a il y a un nœud racine, appelé nom global. Chaque nœud possède une adresse de nœud qui se compose du nom global et d'un ou plusieurs indices (noms des sous-nœuds). a possède les indices b and c; l'adresse de ces noeuds est a->b et a->c.

Les noeuds a->b et a->c->g ont une valeur (d et h), les noeuds a->b->e et a->b->f n'ont aucune valeur. Le noeud a->b possède les indices e et f.

Une description approfondie de cette structure peut être trouvée dans le livre [InterSystems "Utilisation des globaux" \("Using Globals"\)](#).

## Lecture et écriture dans la globale

Native Python API permet de lire et d'écrire directement des données dans la globale IRIS. Le paquet `irisnative` est disponible [sur GitHub](#) — ou si InterSystems IRIS est installé localement sur votre machine, vous le trouverez dans le sous-répertoire `dev/python` de votre répertoire d'installation.

La fonction `irisnative.createConnection` vous permet de créer une connexion à IRIS et la fonction `irisnative.createIris` vous donne un objet de cette connexion avec lequel nous pouvons manipuler la globale. Cet objet possède les méthodes `get` et `set` pour lire/écrire depuis/vers la globale, et une méthode `kill` pour supprimer un nœud et ses sous-nœuds. Il possède également une méthode `isDefined` qui renvoie 0 si le nœud demandé n'existe pas ; 1 s'il a une valeur, mais pas de descendants ; 10 s'il n'a pas de valeur et a des descendants ; ou 11 s'il a une valeur et des 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()
```

Il dispose également d'une méthode `iterator` pour boucler sur les sous-nœuds d'un certain nœud. (L'utilisation sera démontrée dans la section suivante.)

Pour une description complète de chaque méthode, reportez-vous à la documentation de l'interface API .

## Les fichiers de données sur le trafic de San Francisco GTFS

### Stockage des données dans la globale

La Spécification générale des flux de transport en commun (GTFS) est un format pour les horaires et les itinéraires des transports publics.

Regardons comment nous pouvons utiliser l'API native IRIS pour travailler avec [les données GTFS de San Francisco](#) à partir du 10 juin 2019.

Tout d'abord, nous allons stocker les informations des fichiers de données dans la globale IRIS. (Tous les fichiers et toutes les colonnes ne seront pas utilisés dans cette démo). Les fichiers sont au format CSV, où la première ligne indique les noms des colonnes et toutes les autres lignes contiennent les données. En Python, nous commencerons par effectuer les importations nécessaires et établir une connexion à IRIS :

```
import csv
import irisnative

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

Sur la base des noms de colonnes et des données, nous pouvons construire une arborescence judicieuse pour chaque fichier et utiliser `iris.set` pour stocker les données dans la globale.

Commençons par le fichier stops.txt, qui contient tous les arrêts de transport public de la ville. Dans ce fichier, nous n'utiliserons que les colonnes stopid et stopname. Nous allons les stocker dans une globale nommée stops une structure arborescente avec une couche de nœuds, avec les ID des arrêts comme indices et le nom des arrêts comme valeurs des nœuds. Notre structure ressemble donc à stops [stopid]=[stopname]. (Pour cet article, j'utiliserai des crochets pour indiquer quand un indice n'est pas littéral, mais plutôt une valeur lue dans les fichiers de données.)

```
with open("stops.txt", "r") as csvfile:
    reader = csv.reader(csvfile)
    next(reader) # Ignorez les noms des colonnes

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

csv.reader retourne un itérateur de listes qui contiennent les valeurs séparées par des virgules. La première ligne contient les noms des colonnes, nous allons donc la sauter avec next(reader). Nous utiliserons iris.set pour définir le nom de l'arrêt comme valeur de stops -> [stopid].

Ensuite, il y a le fichier routes.txt dont nous utiliserons les colonnes routetype, routeid, routeshortname et routelongname. Une structure globale raisonnable est routes -> [routetype] -> [routeid] -> [routeshortname]=[routelongname]. (Le type d'itinéraire est 0 pour un tram, 3 pour un bus et 5 pour un téléphérique.) Nous pouvons lire le fichier CSV et placer les données dans la globale exactement de la même manière.

```
with open("routes.txt", "r") as csvfile:
    reader = csv.reader(csvfile)
    next(reader) # Ignorez les noms des colonnes

    # 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])
```

Chaque itinéraire a des trajets trips, stockés dans trips.txt, dont nous utiliserons les colonnes routeid, directionid, tripheadsign et tripid. Les trajets sont identifiés de manière unique par leur trip ID (que nous verrons plus tard dans le fichier des temps d'arrêt). Les trajets sur une route peuvent être séparés en deux groupes en fonction de leur direction, et les directions sont associées à des panneaux de tête. Cela conduit à la structure arborescente trips -> [routeid] -> [directionid]=[tripheadsign] -> [tripid].

Nous avons besoin de deux appels iris.set ici — un pour définir la valeur du nœud ID de la direction, et un pour créer le nœud sans valeur de l'ID du trajet.

```
with open("trips.txt", "r") as csvfile:
    reader = csv.reader(csvfile)
    next(reader) # Ignorez les noms des colonnes

    # 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])
```

Enfin, nous allons lire et stocker les temps d'arrêt. Ils sont stockés dans le fichier stoptimes.txt et nous allons utiliser les colonnes stopid, tripid, stopsequence et departuretime. Une première option pourrait consister à utiliser stoptimes -> [stopid] -> [tripid] -> [departuretime] ou si nous voulons conserver la séquence des arrêts, stoptimes -> [stopid] -> [tripid] -> [stopsequence]=[departuretime].

```
with open("stop_times.txt", "r") as csvfile:
    reader = csv.reader(csvfile)
    next(reader) # Ignorez les noms des colonnes

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

## Interrogation des données à l'aide de l'API native

Ensuite, notre objectif est de trouver toutes les heures de départ pour l'arrêt avec le nom donné.

Tout d'abord, nous récupérons l'ID de l'arrêt à partir du nom de l'arrêt donné, puis nous utilisons cet ID pour trouver les heures pertinentes dans le fichier `stoptimes`.

L'appel `iris.iterator("stops")` nous permet d'itérer sur les sous-nœuds du nœud racine `stops`. Nous voulons itérer sur les paires d'indices et de valeurs (pour comparer les valeurs avec le nom donné, et connaître immédiatement l'indice s'il correspond), nous appelons donc `.items()` sur l'itérateur, ce qui définit le type de retour en tuples (indice, valeur). Nous pouvons alors itérer sur tous ces tuples et trouver le bon arrêt.

```
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()
```

Il convient de noter que la recherche d'une clé par sa valeur par itération n'est pas très efficace s'il y a beaucoup de nœuds. Une façon d'éviter cela serait d'avoir un autre tableau, où les indices sont les noms des arrêts et les valeurs sont les IDs. La recherche de la valeur --> clé consisterait alors en une requête dans ce nouveau tableau.

Vous pouvez également utiliser le nom de l'arrêt comme identifiant partout dans votre code au lieu de l'ID de l'arrêt - le nom de l'arrêt est également unique.

Comme vous pouvez le voir, si nous avons une quantité importante d'arrêts, cette recherche peut prendre un certain temps - elle est également connue sous le nom de "balayage complet". Mais nous pouvons profiter des globales et construire le tableau inversé où les noms seront les clés et les IDs les valeurs.

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

stop_id = None

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

En disposant de la globale de `stopnames`, où l'index est le nom et la valeur est l'ID, le code ci-dessus pour trouver

le `stopid` par le nom sera remplacé par le code suivant qui s'exécutera sans une recherche par balayage complet :

```
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()
```

À ce stade, nous pouvons trouver les heures d'arrêt. Le sous-arbre `stoptimes` -> `[stopid]` contient les ID des trajets en tant que sous-nœuds, qui contiennent les temps d'arrêt en tant que sous-nœuds. Nous ne sommes pas intéressés par les ID de trajet - seulement par les temps d'arrêt - donc nous allons itérer sur tous les ID de trajet et collecter tous les temps d'arrêt pour chacun d'eux.

```
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())
```

Nous n'utilisons pas `.items()` sur l'itérateur ici, mais nous utiliserons `.subscripts()` et `.values()` car les ID de trajet sont des `subscripts` (sans valeurs associées) ou la couche inférieure (`[stopsequence]=[departuretime]`), nous sommes seulement intéressés par les valeurs et les heures de départ. L'appel `.update` ajoute tous les éléments de l'itérateur à notre ensemble existant. L'ensemble contient maintenant toutes les heures d'arrêt (uniques) :

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

Rendons les choses un peu plus compliquées. Au lieu de trouver toutes toutes les heures de départ d'un arrêt, nous allons trouver uniquement les heures de départ d'un arrêt pour un itinéraire donné (dans les deux sens) où l'ID de l'itinéraire est donné. Le code permettant de trouver l'ID de l'arrêt à partir du nom de l'arrêt peut être conservé dans son intégralité. Ensuite, tous les ID des arrêts sur l'itinéraire donné seront récupérés.

Le sous-arbre de `trips` -> `[routeid]` est divisé en deux directions, qui ont tous les ID de trajets comme sous-nœuds. Nous pouvons itérer sur les directions comme précédemment, et ajouter tous les sous-nœuds des directions à un ensemble.

```
route = "14334"

selected_trips = set()

directions = iris.iterator("trips", route).subscripts()
pour la direction dans les directions :
    selected_trips.update(iris.iterator("trips", route, direction).subscripts())
```

L'étape suivante consiste à trouver les valeurs de tous les sous-nœuds de `stoptimes` -> `[stopid]` -> `[tripid]` où `[stopid]` est l'identifiant de l'arrêt récupéré et `[tripid]` est l'un des identifiants de trajet sélectionnés. Nous itérons sur l'ensemble `selectedtrips` pour trouver toutes les valeurs pertinentes :

```
all_stop_times = set()

pour le trajet dans selected_trips:
    all_stop_times.update(iris.iterator("stoptimes", stop_id, trip).values())
```

```
pour stop_time dans sorted(all_stop_times):  
    print(stop_time)
```

Un dernier exemple montre l'utilisation de la fonction `isDefined`. Nous allons développer le code écrit précédemment : au lieu de coder en dur l'ID de la route, le nom court d'une route est donné, puis l'ID de la route doit être récupéré sur cette base. Les noeuds avec les noms de route sont sur la couche inférieure de l'arbre. La couche supérieure contient les ID des routes. Nous pouvons itérer sur tous les types de route, puis sur tous les ID de route, et si le noeud routes -> [routetype] -> [routeid] -> [routeshortname] existe et a une valeur (`isDefined` retourne 1), alors nous savons que [routeid] est l'ID que nous recherchons.

```
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()
```

Ce code sert à remplacer la ligne `route = "14334"` codée en dur.

Lorsque toutes les opérations IRIS sont terminées, nous pouvons fermer la connexion à la base de données :

```
conn.close()
```

## Prochaines étapes

Nous avons couvert comment l'API native de Python peut être utilisée pour accéder à la structure de données d'InterSystems IRIS, puis être appliquée aux données des transports publics de San Francisco. Pour une plongée plus profonde dans l'API, vous pouvez consulter la documentation . L'API native est également disponible pour [Java](#), [.NET](#) et [Node.js](#).

[#API](#) [#Python](#) [#InterSystems](#) [IRIS](#)

---

URL de la source: <https://fr.community.intersystems.com/post/iris-native-api-pour-python>