Données géolocalisées et visualisation cartographique avec Python

Joeffrey Drouard

Version de Python et Modules Python

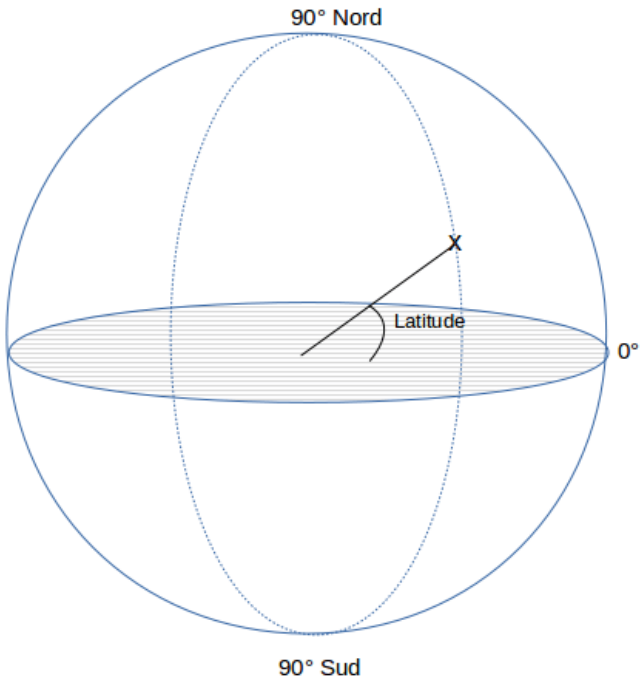
- python 3.5
- folium 0.5.0
- pandas
- shapely 1.6.4

Quelques notions

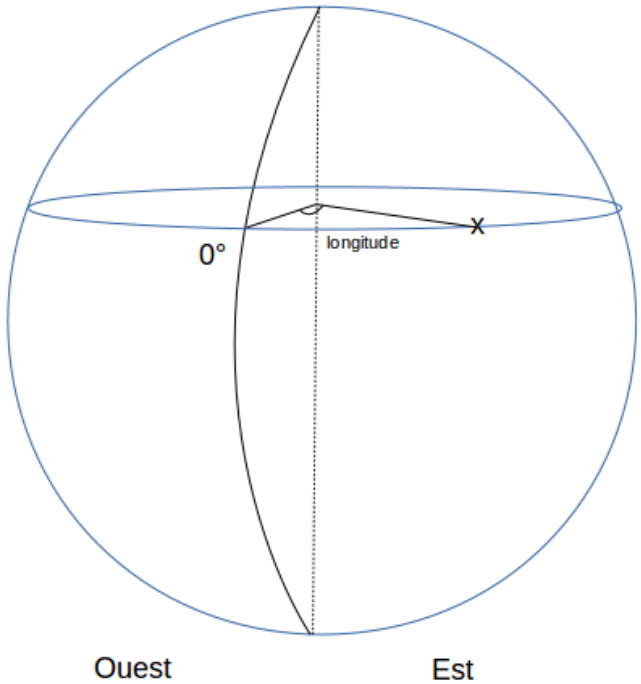
La longitude et latitude permettent de donner une adresse à tous les lieux de la Terre. Il s'agit donc de repérer un endroit sur la sphère terrestre. Un point se repère par la distance à l'équateur - latitude - et la distance à un Méridien de référence (en général Greenwitch) - longitude.

Mais ce qu'on appelle distance n'en est pas une en kilomètres. Ces données sont en degrés (°): un degré mesure l'écartement d'un angle, comme par exemple l'angle droit est de 90° dans un triangle rectangle.

La **latitude** d'un point représente l'angle entre le plan de l'équateur et une droite passant par le centre de la terre et l'endroit du point à repérer. Cet angle varie entre 0 et 90° et on précise s'il s'agit de l'hémisphére Nord ou Sud.



La **longitude** d'un point représente sa position Est-Ouest. Il s'agit de l'angle formé avec le méridien de référence (en général Greenwitch). Il s'agit d'un angle entre 0 et 180°, avec l'indication Est ou Ouest selon qu'on se trouve à droite ou à gauche du méridien de référence.



Il y a deux façons d'écrire les degrés, plus particulièrement les sous-degrés, c'est à dire ce qui vient après la virgule.

1° représente 1/360 d'un tour complet, c'est comme découper un gâteau en 360 parts égales, 1° est l'angle formé par la pointe d'une part.

Mais plutôt que de diviser 1° en 10 (**système décimal**), on fait comme pour les heures, on les divise en 60, et on appelle les unités des minutes d'arc (plutôt que des minutes, qui sont des unités de temps). Cette idée originale vient des Babyloniens qui comptaient en base 60 et non comme nous en base 10. Après les minutes d'arc, on divise encore par 60 pour obtenir les secondes d'arc. On appelle ce système **DMS** pour Degrees, Minutes, Seconds.

Dans le système **DMS**, les coordonnées de la Tour Eiffel sont 48° 51' 29" N, 2° 17' 40" E

On peut cependant transformer les sous-degrés en sous-degrés décimaux pour que les choses soient plus simples. Comme 1h30 = 1.5h, on écrit pour la Tour Eiffel: 48.8581° N, 2.2944° E dans le **système décimal**.

- Dans le système décimal, on peut également remplacer N/S et E/W en précédant les coordonnées d'un éventuel signe moins. Pour Rio de Janeiro, on écrit 22.9098 S, 43.2094 W ou -22.9098, -43.2094

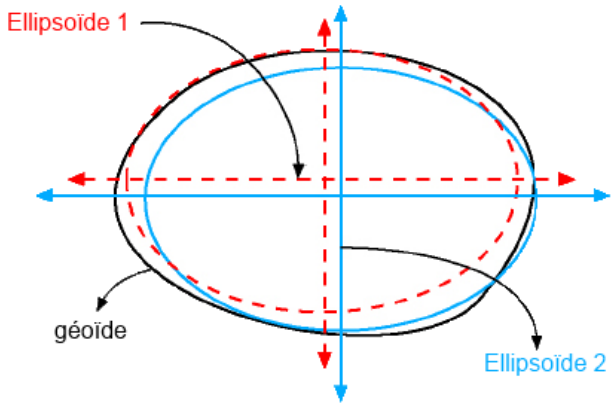
La Terre n'est pas une sphère, elle est aplatie aux pôles et surtout n'a pas une forme régulière, c'est en fait un géoïde (ballon cabossé)

Mais un géoïde est une forme géométrique trop complexe pour permettre des calculs performants (e.g., difficile de calculer des distances...), et donc pour être utilisée comme représentation de la Terre dans un système d'information géographique.

Solution $\Rightarrow$ remplacer la véritable surface du géoïde par une surface lisse et régulière: un ellipsoïde... Mais quel ellipsoïde adopté?

Chaque système géodésique (datum) est caractérisé par le choix

- d'un modèle d'ellipsoïde
- d'une position du centre de la Terre
- d'une unité de mesure
- d'un méridien de référence



$\Rightarrow$ Les coordonnées géographiques n'ont aucun sens si on ne les accompagne pas des informations sur le système géodésique dans lequel elles sont exprimées.

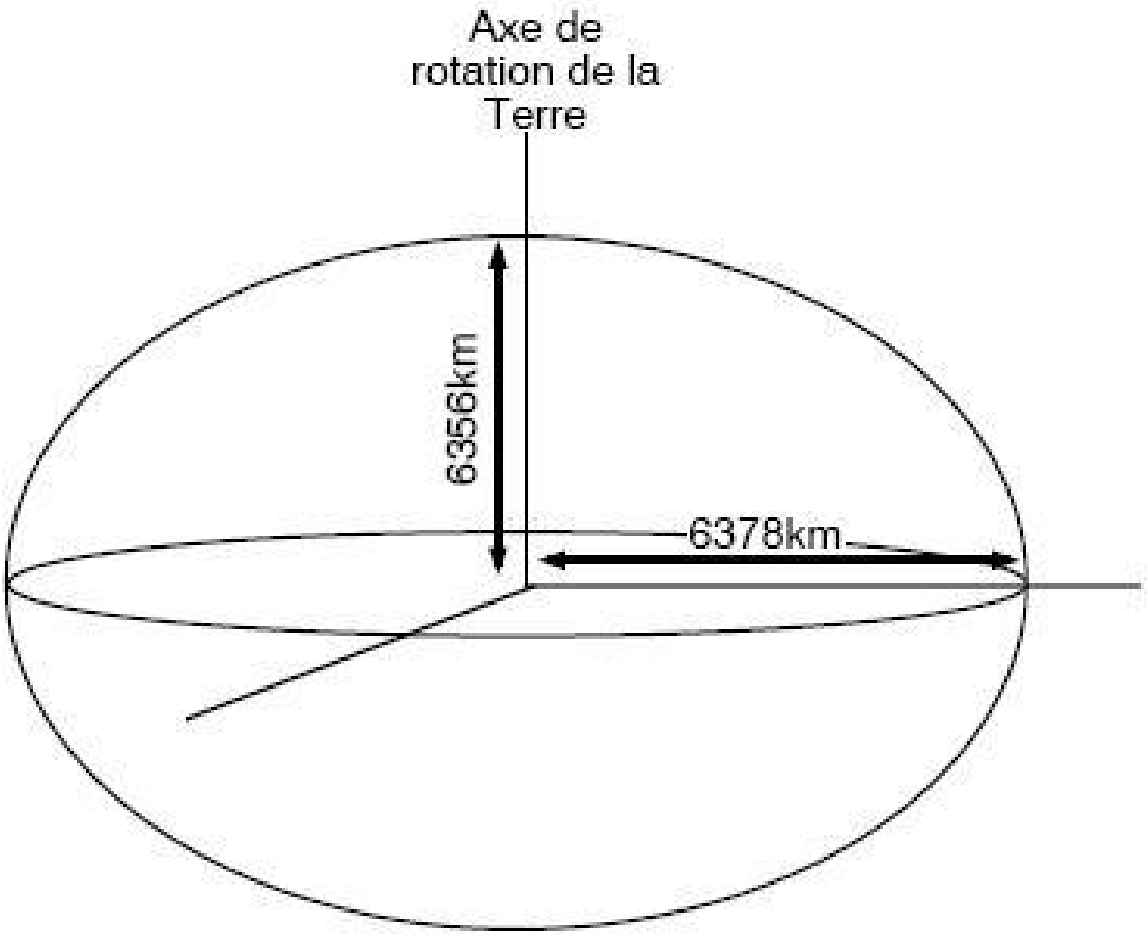
Système global

Lorsqu'on a besoin de cordonnées à l'échelle de la Terre entière, on va devoir choisir un système de coordonnées basé sur un ellipsoïde qui épouse au mieux la forme de la Terre.

Le système géodésique **WGS84** est devenu la référence des systèmes géocentriques. Il est utilisé à peu près partout : Applications GPS, Google Maps, Bing Maps, OpenStreetMaps... (rmq: 4326 is just the EPSG identifier of WGS84)

Sa précision est en moyenne de 1 à 2 mètres

Le système géodésique WGS84



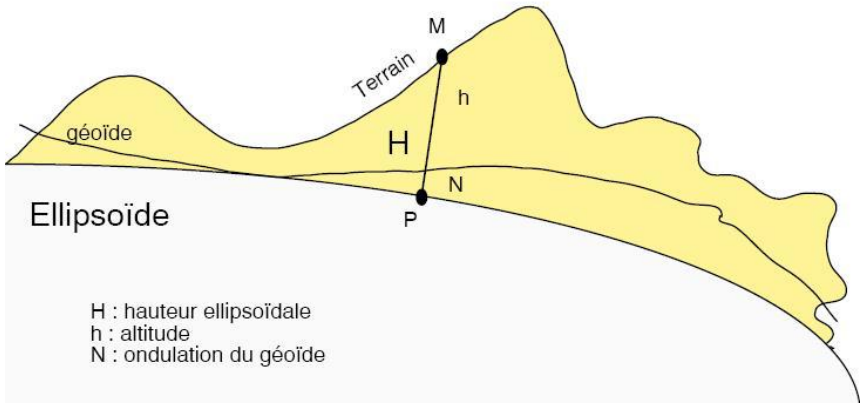
Système local

Lorsqu'on veut travailler avec une zone particulière du globe (un continent, un pays, ...), on sélectionne un système géodésique local qui permet de faire correspondre au mieux la surface de l'ellipsoïde avec la surface de la terre.

Le système géodésique local est beaucoup plus précis qu'un système géocentrique global dans la zone pour laquelle il a été défini.

Exemple: en Amérique du Nord on peut utiliser Le NAD83 (pour North American Datum)

Les altitudes sont mesurées par rapport à la surface du géoïde (et non par rapport à l'ellipsoïde). Pour les connaître il faut les mesurer sur place.



Sur la figure il faut comprendre que $H = h+N$. L'altitude correspond à h.

Créer une carte avec le module Folium

Folium est un module qui permet de générer des cartes interactives en s'appuyant sur leaflet, une librairie JavaScript.

Pour utiliser Folium, les coordonnées des éléments géométriques doivent être dans le système de projection WGS84 (cf pour changer de système de projection avec Python).

Créer une carte avec le module Folium

Représentons une carte de Rio de Janeiro.

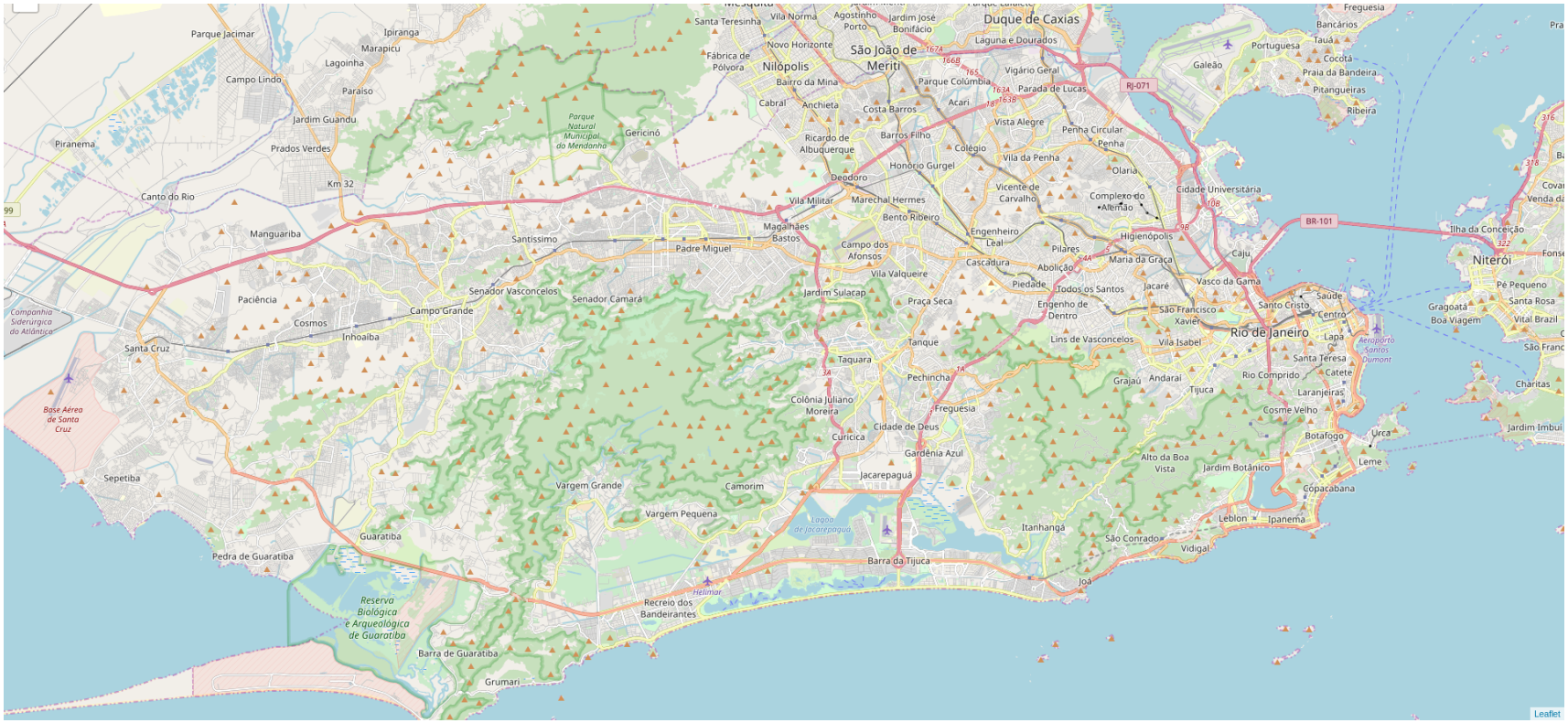
```
import folium
## Création de la carte
lat=-22.9133
lon=-43.4146
macarte = folium.Map(location=[lat,lon], zoom_start=12)

## Définir l'arrière plan de la carte
folium.TileLayer("stamenterrain").add_to(macarte)

## Enregistrer la carte - vérifier que le dossier Carte existe
macarte.save('Carte/ma_premiere_carte_vide.html')
```

TileLayer permet de choisir un l'arrière plan de la carte (openstreetmap, stamenterrain, Mapbox Control Room, stamenwatercolor, stamen toner...) - peut également être passé comme un argument dans *folium.Map*

```
macarte = folium.Map(location=[lat,lon], zoom_start=5, tiles = "stamenterrain")
```

cf exemple1.py

Ajouter des icônes



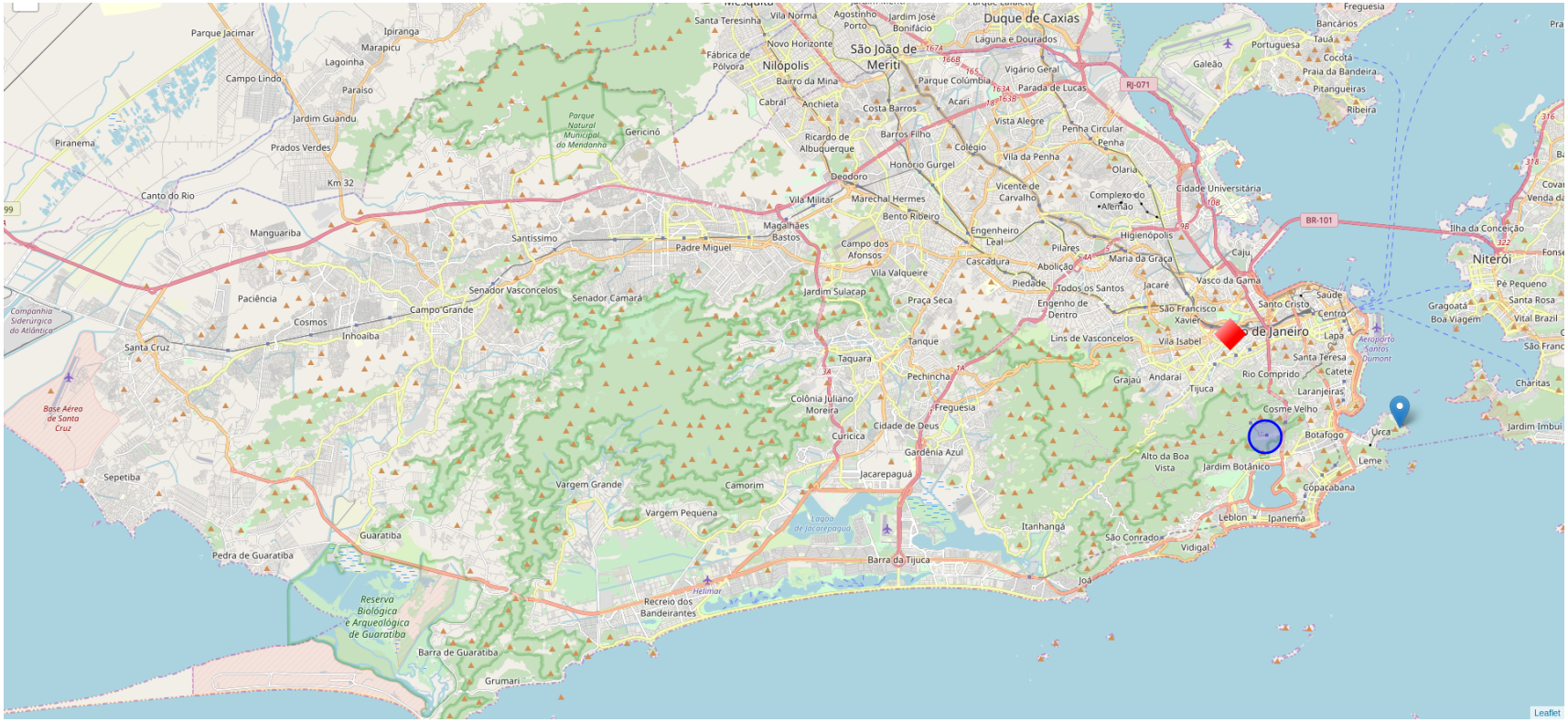
Ajouter des icônes

Ajoutons à notre carte, 3 icônes qui indiquent l'emplacement du Maracana (lat=-22.9127,lon=-43.2261), du Pain de Sucre (lat=-22.9492,lon=-43.1545) et du Corcovado (lat=-22.9523,lon=-43.4114).

```
# le Maracana - carré rouge
folium.RegularPolygonMarker([-22.9127,-43.2261],color='none',
        fill_color='red',number_of_sides=4,radius=5,popup='Maracana')
        .add_to(macarte)

# Pain de sucre -  icône similaire à celle de google map
folium.Marker([-22.9492,-43.1545],popup='Pain de Sucre').add_to(macarte)

# Corcovado - cercle bleu
folium.CircleMarker([-22.9523,-43.4114],color='blue',fill='True',
        fill_color='blue',radius=10,popup='Corcovado')
        .add_to(macarte)

macarte.save('Carte/une_carte_avec_marqueurs.html')
```

cf exemple2.py

Ajouter des icônes



- *color* la couleur du contour des icônes. Mettre *none* si on ne souhaite pas de contour

- *fill_color* la couleur de l'icône. Pour les arguments (*Color* et *fill_color*), il faut choisir soit une couleur (blue, red, silver...) soit un code couleur. Pour plus d'informations sur les couleurs dans python cf code couleur

- *opacity* pour avoir des couleurs plus ou moins foncées ou transparentes ($= \{0.1, 0.2, ...1\}$)

- *fill_opacity* pour avoir des contours plus ou moins foncés ou transparents ($= \{0.1, 0.2, ...1\}$)

- *number_of_sides* permet de définir le nombre de côtés de l'icône - argument à utiliser uniquement avec RegularPolygonMarker ($= \{3, 4, ...\}$)
- *radius* permet de définir la taille de l'icône ($= \{1, 2, 3, ...\}$)
- *popup* permet d'associer un commentaire à l'icône.

Ajouter des couches

On va utiliser la méthode *choropleth*. Cette méthode permet de représenter des couches à partir de données au format GeoJSON.

GeoJSON (de l'anglais Geographic JSON) est un format ouvert d'encodage d'ensemble de données géospatiales simples utilisant la norme JSON (JavaScript Object Notation).

Le format GeoJSON permet de décrire des données de type point, ligne, chaîne de caractères, polygone, ainsi que des ensembles et sous-ensembles de ces types de données et d'y ajouter des attributs d'information qui ne sont pas spatiales.

Un objet GeoJSON est un dictionnaire avec au moins une clé *features*. Cette clé renvoie à une liste où chaque élément correspond à une couche.

```
Base_geo={}
Base_geo['features']=[]
```

On va enregistrer différentes couches dans la liste *Base_geo['features']*. Chaque élément de la liste correspond à une couche. La longueur de la liste *Base_geo['features']* est donc égale au nombre de couche.

Exemple

On va créer deux couches qui seront des carrés, de longueur 1°.

- Couche 1 centrée sur lat=1.5 et lon=1.5
- Couche 2 centrée sur lat=-0.5 et lon=1.5

Remarques:

- Les couches vont être de la forme [[lon1,lat1],[lon2,lat2]...,[lon1,lat1]]; une liste de listes avec pour chaque liste la **longitude** en premier élément et la **latitude** en second élément (contrairement à *folium.Map*)
- Pour les polygones, le premier élément et le dernier élément de la liste sont identiques (pas pour les segments)

Ajouter des couches
Créer des couches

```
## Creation de notre base GeoJSON
Base_geo={}
Base_geo['features']=[]

# Couche 1
feature1={}
feature1['type']='Feature'
feature1['geometry']={ "type": "Polygon", "coordinates":
        [
        [[2,2], [2,1], [1,1],[1,2],[2,2]]
        ]
}
Base_geo['features'].append(feature1)

# Couche 2
feature2={}
feature2['type']='Feature'
feature2['geometry']={ "type": "Polygon", "coordinates":
        [
        [[2,0], [2,-1], [1,-1],[1,0],[2,0]]
        ]
}
Base_geo['features'].append(feature2)
```

```
## Création de la carte avec les différentes couches
macarte = folium.Map(location=[0,0], zoom_start=5)
macarte.choropleth(Base_geo)
macarte.save('Carte/carte_avec_couches.html')
```

cf exemple3.py

Comment enregistrer nos données geojson et les lire?

```
## Sauvegarde de notre base GeoJSON - vérifier que le dossier Data existe
with open('Data/maBaseGeoJSON.geojson','w') as output:
        json.dump(Base_geo,output)

## Charger notre base - vérifier que le dossier Data existe
with open('Data/maBaseGeoJSON.geojson','r') as output:
        Base_geo=json.load(output)
```

Problème: ici nos couches n'ont aucun attribut

- on pourrait associer à chaque couche un attribut (e.g., le pib, la température moyenne, le taux de chômage...)
- avec des couleurs différentes en fonction du niveau/de la valeur de cet attribut.

Pour représenter les couches avec des couleurs différentes en fonction de la valeur de leurs attributs, on a besoin:

i. Des attributs au format dataframe

   Un dataframe se comporte comme un dictionnaire dont les clés sont les noms des colonnes et les valeurs sont des séries.

ii. Une clé qui relie nos attributs (enregistrés dans le dataframe) aux identifiants des couches de notre base GeoJSON

i. Ajoutons une clé à notre base GeoJSON qui permet d'identifier les couches

```
feature1['properties']={}
feature1['properties']['id']='Zone1'

feature2['properties']={}
feature2['properties']['id']='Zone2'
```

ii. Enregistrons nos attributs dans un dataframe

```
df_attribut=pandas.DataFrame({'Zone': ['Zone1','Zone2'],
        'Attribut 1': [0,1], 'Attribut 2': [50,56]})
```

df_attribut est une base de données au format dataframe avec 3 colonnes *Zone*, *Attribut 1* et *Attribut 2*. Il est possible de relier cette base avec les données GeoJSON en utilisant les clés respectives *Zone* et *identifiant*.

Ajouter des couches
Ajouter des attributs

```
## Création de la carte avec les différentes couches & différents attributs
macarte = folium.Map(location=[0,0], zoom_start=6)
macarte.choropleth(Base_geo,data=df_attribut,columns=('Zone', 'Attribut 1')
        ,key_on='feature.properties.id', fill_color='BuPu'
        ,legend_name='ATTRIBUT 1')
macarte.save('Carte/carte_avec_couches_attributs.html')
```

cf exemple4.py

Dans la méthode choropleth, on a rajouté les arguments suivants:

- *data* : la base de données (au format dataframe) qui contient les attributs
- *columns* : le premier élément renvoie à la clé qui permet de relier les attributs aux données GeoJSON, le second élément renvoie aux valeurs de l'attribut
- *key_on* : indique le chemin de la clé dans les données GeoJSON
- *fill_color* : la couleur utilisée pour représenter les couches (et les variations de l'attribut associé à ces couches) - on peut utiliser un "hex" code, un nom de couleur ou la palette de couleurs BuGn, BuPu, GnBu, OrRd, PuBu, PuBuGn, PuRd, RdPu, YlGn, YlGnBu, YlOrBr, YlOrRd.
- *legend_name* : titre associé à l'attribut.

Dans un objet GeoJSON, Les couches peuvent être de différents formats: point, segment, polygone, polygone à trous, multipolygones...

- cf Lien wikipedia GeoJSON pour les différents formats

Le segment

```
Base_geo={}
Base_geo['type']='FeatureCollection'
Base_geo['features']=[]

feature={}
feature['type']='Feature'
feature['geometry']={}
feature['geometry']["type"]="LineString"
feature['geometry']['coordinates']=[
[15, 5],[5, 15],[20, 20]
                                    ]

Base_geo['features'].append(feature)

macarte = folium.Map(location=[10,10], zoom_start=6)
macarte.choropleth(Base_geo)
macarte.save('Carte/carte_segment.html')
```

Polygone sans trou

```
Base_geo={}
Base_geo['type']='FeatureCollection'
Base_geo['features']=[]

feature={}
feature['type']='Feature'
feature['geometry']={}
feature['geometry']['type']='Polygon'
feature['geometry']['coordinates']=[
[[2,2], [2,1], [1,1],[1,2],[2,2]]
                                    ]

Base_geo['features'].append(feature)

macarte = folium.Map(location=[2,2], zoom_start=6)
macarte.choropleth(Base_geo)
macarte.save('Carte/carte_polygone.html')
```

[
[[lo,la],[lo,la],...,[lo,la]]
]

Polygone à trous (ici 2)

```
Base_geo={}
Base_geo['type']='FeatureCollection'
Base_geo['features']=[]

feature={}
feature['type']='Feature'
feature['geometry']={}
feature['geometry']['type']='Polygon'
feature['geometry']['coordinates']=[
[[2,2], [2,1], [1,1],[1,2],[2,2]],
[[1.75,1.75], [1.75,1.5], [1.5,1.5],[1.5,1.75],[1.75,1.75]],
[[1.25,1.25], [1.25,1.0], [1.0,1.0],[1.0,1.25],[1.25,1.25]]
                                    ]

Base_geo['features'].append(feature)

macarte = folium.Map(location=[2,2], zoom_start=6)
macarte.choropleth(Base_geo)
macarte.save('carte_polygone_a_trou.html')
```

[
[[lo,la],[lo,la],...,[lo,la]], [[lo,la],[lo,la],...,[lo,la]],[[lo,la],[lo,la],...,[lo,la]]
]

Multipolygone sans trou

```
...
feature={}
feature['type']='Feature'
feature['geometry']={}
feature['geometry']['type']='MultiPolygon'
feature['geometry']['coordinates']=[
[[[-1,-1], [-1,-3], [-4,-3],[-4,-1],[-1,-1]]],
[[[4,2], [4,1], [3,1],[3,2],[4,2]]]
                                  ]

Base_geo['features'].append(feature)

macarte = folium.Map(location=[0,0], zoom_start=6)
macarte.choropleth(Base_geo)
macarte.save('carte_multipolygone.html')
```

[
[[[lo,la],[lo,la],...,[lo,la]]],
[[[lo,la],[lo,la],...,[lo,la]]]
]

Multipolygone à trous

```
...
feature={}
feature['type']='Feature'
feature['geometry']={}
feature['geometry']['type']='MultiPolygon'
feature['geometry']['coordinates']=[
[[[-1,-1], [-1,-3], [-4,-3],[-4,-1],[-1,-1]],
[[-1.5,-1.5],[-1.5,-2.5],[-2.5,-2.5],[-2.5,-1.5],[-1.5,-1.5]]],
[[[4,2], [4,1], [3,1],[3,2],[4,2]]]
                                    ]

Base_geo['features'].append(feature)
...
macarte.save('carte_multipolygone_a_trou.html')
```

[
[[[lo,la],[lo,la],...,[lo,la]],[[lo,la],[lo,la],...,[lo,la]]],
[[[lo,la],[lo,la],...,[lo,la]]]
]

cf exemple5.py

Le module shapely permet d'analyser des géométries dans un plan cartésien. Par exemple, il est possible:

- Définir le centroid d'un polygone avec Shapely
- Définir si un point appartient ou pas à un polygone avec Shapely

Définir le centroid d'un polygone avec Shapely

```
from shapely.geometry.polygon import Polygon

L=[[0,0],[0,1],[-1,1],[-1,0],[0,0]]
mon_polygone=Polygon(L)

## Définir le centroid
# x_centroid et y_centroid est le centre du polygone
point_centroid=mon_polygone.centroid
x_centroid=point_centroid.x
y_centroid=point_centroid.y
print(x_centroid,y_centroid)
```

Définir si un point appartient ou pas à un polygone avec Shapely

```
from shapely.geometry import Point
from shapely.geometry.polygon import Polygon

L=[[0,0],[0,1],[-1,1],[-1,0],[0,0]]
mon_polygone=Polygon(L)

point1=Point(-1.5,0.5) # n'appartient pas au polygone
point2=Point(0.5,0.5) # n'appartient pas au polygone
point3=Point(-0.1,0.5) # appartient au polygone

# renvoie True ou False selon que le point appartient au polygone L ou pas
# Attention : renvoie False si le point est sur la frontière
print(mon_polygone.contains(point1))
print(mon_polygone.contains(point2))
print(mon_polygone.contains(point3))
```

Attention: si le point est sur la frontière du polygone, la méthode *contains* renvoie *False*. cf exemple6.py

Le module geopy permet de:

- Géocoder des adresses par le biais du service Nominatim d'OpenStreetMaps (besoin d'une connection internet)
- Calculer la distance entre deux points

Géocoder des adresses

```
from geopy.geocoders import Nominatim

## Trouver les coordonnées d'une adresse
geolocator = Nominatim()
location=geolocator.geocode('rue de Belleville, Paris')
print(location.address)
G1=(location.latitude, location.longitude)

location=geolocator.geocode('place Hoche, Rennes')
print(location.address)
G2=(location.latitude, location.longitude)
```

Calculer la distance entre deux points

```
from geopy.distance import great_circle
from geopy.distance import vincenty

## Calculer la distance entre G1 et G2
print(great_circle(G1, G2).meters)
print(vincenty(G1, G2).meters)
```

Remarques

- *Vincenty* utilise par défaut le système géodésique WGS84 (possibilité de le modifier)
- *great-circle* représente la terre sous forme de sphère avec un radius de 6371 km

$\Rightarrow$ Le module *Vincenty* est plus lent que le module *great-circle*, toutefois il est plus précis.

cf exemple7.py

Exercice 1

Vous disposez d'une base de données BaseExercice.geojson (il y a 3 couches dans cette base: P1, P2, P3).

Créer une carte avec ces couches et représenter les points suivants (longitude,latitude):

- point1=(4.5,4.5), point2=(6.2,6.5), point3=(1.9,1.7), point4=(1.7,1.7), point5=(-1.2,-1.2), point6=(-1.6,-2), point7=(3.5,1.5)

avec les couleurs suivantes: jaune si dans P1, rouge si dans P2, vert si dans P3 et noir sinon

Correction de l'exercice cf correction1.py

Exercice 2

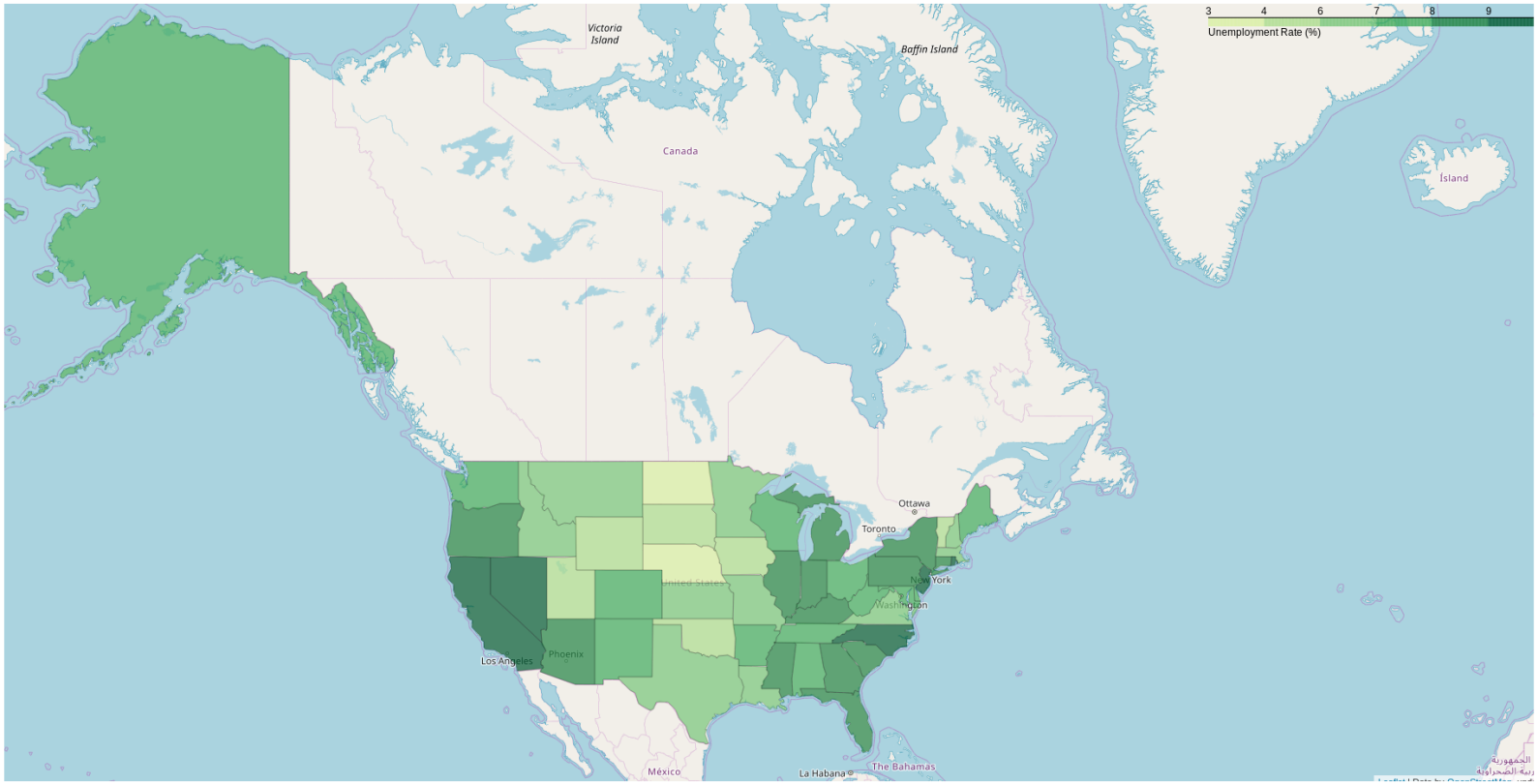
Vous disposez de deux bases de données:

- une base de données us-states.json avec les frontières des états américains
- une base de données US_Unemployment_Oct2012 avec le taux de chômage dans chaque état en octobre 2012

Créer une carte représentant les états américains avec des couleurs plus ou moins foncées en fonction du taux de chômage.

Correction de l'exercice cf correction2.py

Exercice 2


3
4
6
7
8
9
Unemployment Rate (%)
Victoria Island
Baffin Island
Canada
Ísland
Ottawa
Toronto
New York
Washington
Phoenix
Los Angeles
México
La Habana
The Bahamas

Formats de données géolocalisées

Il existe une multitude de formats pour les données géolocalisées:

- Geojson
- Shapefile
- KML
- ASCII
- NetCDF
- ...

Le format shapefile

Le shapefile, ou « fichier de formes » est un format de fichier pour les systèmes d'informations géographiques (SIG). Initialement développé par ESRI pour ses logiciels commerciaux, ce format est désormais devenu un standard, et est utilisé par un grand nombre de logiciels libres (MapServer, Grass, Udig, MapGuide OpenSource GmapCreator, QGIS, GvSIG, etc.).

Il contient toute l'information liée à la géométrie des objets, qui peuvent être:

- des points
- des lignes
- des polygones
  - le cercle n'est pas supporté.

Le format shapefile

Son extension est classiquement SHP, et il est toujours accompagné de deux autres fichiers de même nom, et d'extensions :

- un fichier DBF, qui contient les données attributaires relatives aux objets contenus dans le shapefile
- un fichier SHX, qui stocke l'index de la géométrie

Sans ces deux autres fichiers, on ne peut rien faire ! D'autres fichiers peuvent être également fournis: .sbn et .sbx (index spatial des formes), .fbn et .fbx (index spatial des formes pour les shapefile en lecture seule) ...

Pour transformer un fichier (ou plutôt les fichiers) shapefile en geojson cf convertSHP-GEOJSON.py

Le format KML

KML (auparavant connu sous le nom de Keyhole Markup Language) est un format de fichier XML qui permet d'afficher des informations dans un contexte géographique. Les informations KML peuvent s'afficher dans de nombreux navigateurs terrestres, notamment ArcGIS Earth et ArcGIS Pro.

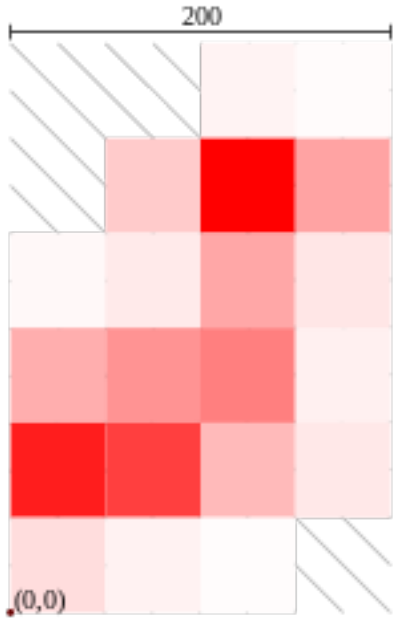
Le fichier KML est la source de la couche dans la carte ou scène. Vous pouvez utiliser un fichier .kml ou .kmz (compressé) ou une URL qui désigne un fichier KML comme source de données.

Aide pour convertir un fichier KML en Geojson

Le format ASCII grid

	25	75	125	175
275	NA	NA	5	2
225	NA	20	100	36
175	3	8	35	10
125	32	42	50	6
75	88	75	27	9
25	13	5	1	NA

Supposons qu'on souhaite enregistrer les valeurs associées à 24 cellules (6*4) de longueur 50 unités. E.g., la valeur associée au carré de centroid (25,25) est de 13.



Le format ASCII grid

Il y a 3 types d'informations:

i. la taille des cellules (50 unités)

ii. le nombre de cellules (6*4)

iii. les valeurs des cellules

Le format ASCII grid format permet d'enregistrer les données (en 3 dimensions) à moindre coût mémoire.

La structure du fichier ASCII se compose d'informations d'en-tête contenant un ensemble d'informations sur les cellules (i. et ii.), suivies de valeurs liées aux cellules (iii.).

Le format ASCII grid

Le fichier .asc sera de la forme suivante:

---

ncols 4
nrows 6
xllcorner 0.0
yllcorner 0.0
cellsize 50.0
NODATA_value -9999
-9999 -9999 5 2
-9999 20 100 36
3 8 35 10
32 42 50 6
88 75 27 9
13 5 1 -9999

Le format ASCII grid

Comment interpréter le fichier .asc?

- *ncols* et *nrows* correspondent au nombre de colonnes et de lignes dans le plan défini par le fichier ASCII.
- *xllcorner* et *yllcorner* sont les coordonnées du coin inférieur gauche de la cellule inférieure gauche. (on peut également utiliser *xllcenter* et *yllcenter* pour spécifier l'origine par les coordonnées du centre de la cellule inférieure gauche)
- *cellsize* est la taille de cellule (la longueur)
- *NODATA_value* est la valeur qui doit représenter les cellules NoData. Ici -9999 signifie que la valeur de la cellule n'est pas observée.
- A partir de la septième ligne on a les valeurs associées à chaque cellule. La valeur *13* correspond à la cellule en bas à gauche.

Le format ASCII grid

Ce format est composé de plusieurs fichiers. Le fichier central .asc contient les données. Il y a d'autres fichiers (readme.txt, .prj) qui peuvent fournir des informations complémentaires.

- Exemple: *the data distributed here are in ARC GRID, ARC ASCII and Geotiff format, in decimal degrees and datum WGS84* $\Rightarrow$ permet de savoir que les données sont des coordonnées géographiques dans un système décimal

Ces données géolocalisées sous forme de "grid" (cellules) utilisent souvent des niveaux de résolution en arc-minute

- 60 arc-minute (1 degré)
- 30 arc-minute (0.5 degré)
- 5 arc-minute (0.083333333... degré)

Le format NetCDF

Le format NetCDF (Network Common Data Form) est couramment utilisé dans des applications de climatologie, de météorologie et d'océanographie.

L'extension est .nc ou .cdf

Le format de données est « auto-documenté » c'est-à-dire qu'il existe un en-tête qui décrit la disposition des données dans le reste du fichier, et en particulier des tableaux de données.

Cet en-tête contient aussi une liste arbitraire de métadonnées se présentant sous la forme d'attribut de type nom/valeur.

Données de cellules de 0.5*0.5 de 1901 à 2016. Obtenues sur crudata.uea.ac.uk - cf DataClimat.

Ouvrir la base

```
dataset =Dataset('Data/DataClimat/temp_max_1901-2016.dat.nc','r')
```

Comprendre les informations de la base

```
>>> dataset.variables.keys()
odict_keys(['lon', 'lat', 'time', 'tmx', 'stn'])
```

Il y a 5 clés qui sont *lon*, *lat*, *time*, *tmx*, *stn*.

```
>>> dataset.variables['time']

<class 'netCDF4._netCDF4.Variable'>
float32 time(time)
    long_name: time
    units: days since 1900-1-1
    calendar: gregorian
unlimited dimensions: time
current shape = (1392,)
filling off
```

Les informations sur les unités de temps sont en *jours* et commence le 1er janvier 1900. La dimension est de 1392, ce qui correspond au nombre de mois entre janvier 1901 et décembre 2016.

```
>>> dataset.variables['lon']

<class 'netCDF4._netCDF4.Variable'>
float32 lon(lon)
    long_name: longitude
    units: degrees_east
unlimited dimensions:
current shape = (720,)
filling off
e = (1392,)
filling off
```

Les informations sur les longitudes ont une dimension de 720 ce qui correspond à des demi degrés (il y a 360 degrés de -180 à 180)

```
>>> dataset.variables['lat']

<class 'netCDF4._netCDF4.Variable'>
float32 lat(lat)
    long_name: latitude
    units: degrees_north
unlimited dimensions:
current shape = (360,)
filling off
```

Les informations sur les latitudes ont une dimension de 360 ce qui correspond à des demi degrés (il y a 180 degrés de -90 à 90)

```
>>> dataset.variables['tmx']

<class 'netCDF4._netCDF4.Variable'>
float32 tmx(time, lat, lon)
    long_name: near-surface temperature maximum
    units: degrees Celsius
    correlation_decay_distance: 1200.0
    _FillValue: 9.96921e+36
    missing_value: 9.96921e+36
unlimited dimensions: time
current shape = (1392, 360, 720)
filling off
```

Les informations sur les températures ont une dimension de (1392, 360, 720) - ce qui correspond à (date,lat,lon).

```
>>> dataset['time'][0:1392].data
array([  380.,   410.,   439., ..., 42657., 42688., 42718.], dtype=float32)
```

La première valeur associée à la dimension temps correspond au 380ème jour depuis le 1er janvier 1900 (i.e., 16 janvier 1901), la seconde valeur associée à la dimension temps correspond au 410ème depuis le 1er janvier 1900 (i.e., 15 février 1901)... la 1392ème valeur associée à la dimension temps correspond au 42718ème jour depuis le 1er janvier 1900 (i.e., 16 décembre 2016).

```
>>> from datetime import date
>>> from datetime import timedelta
>>> date(1900,1,1)+timedelta(days=42718)
2016-12-16
```

```
>>> dataset['lat'][0:360].data

array([-89.75, -89.25, -88.75, -88.25, -87.75, -87.25, -86.75, -86.25,
       ...
        86.25,  86.75,  87.25,  87.75,  88.25,  88.75,  89.25,  89.75],
      dtype=float32)
```

La première valeur associée à la dimension latitude correspond à la latitude -89.75 ... la 360ème valeur associée à la dimension latitude correspond à la latitude 89.75.

```
>>> dataset['lon'][0:720].data

array([-179.75, -179.25, -178.75, -178.25, -177.75, -177.25, -176.75,
         ...
        177.25,  177.75,  178.25,  178.75,  179.25,  179.75],
      dtype=float32)
```

La première valeur associée à la dimension longitude correspond à la longitude -179.75 ... la 720ème valeur associée à la dimension longitude correspond à la longitude 179.75.

Comment trouver la température maximale en février 1984 dans la grille qui contient Rio de Janeiro (-22.9098, -43.2094)?

```
>>> dataset['lat'][134].data
array(-22.75, dtype=float32)

>>> dataset['lon'][273].data
array(-43.25, dtype=float32)

>>> dataset['time'][1000].data
array(30816., dtype=float32)

>>> date(1900,1,1)+timedelta(days=30816)
datetime.date(1984, 5, 16)
```

La dimension à retenir est (1000,134,273).

```
>>> dataset['tmx'][1000].tolist()[134][273]
27.899999618530273
```

Où trouver des données géolocalisées?

grid.prio.org est un portail de données opensource. Ce site propose des données géolocalisées à un niveau assez fin : "a standardized spatial grid structure with global coverage at a resolution of 0.5 x 0.5 decimal degrees".

- Pour obtenir les coordonnées des cellules, il faut télécharger PRIO-GRID cell shapefile à partir de ce lien. Les informations sont offertes dans un format shp.

Arcgis est une plateforme qui offre l'accès à des données géolocalisées.

- Consulter ce lien pour avoir des informations plus précises sur les requêtes sur arcgis (ou cette vidéo pour un tutoriel).