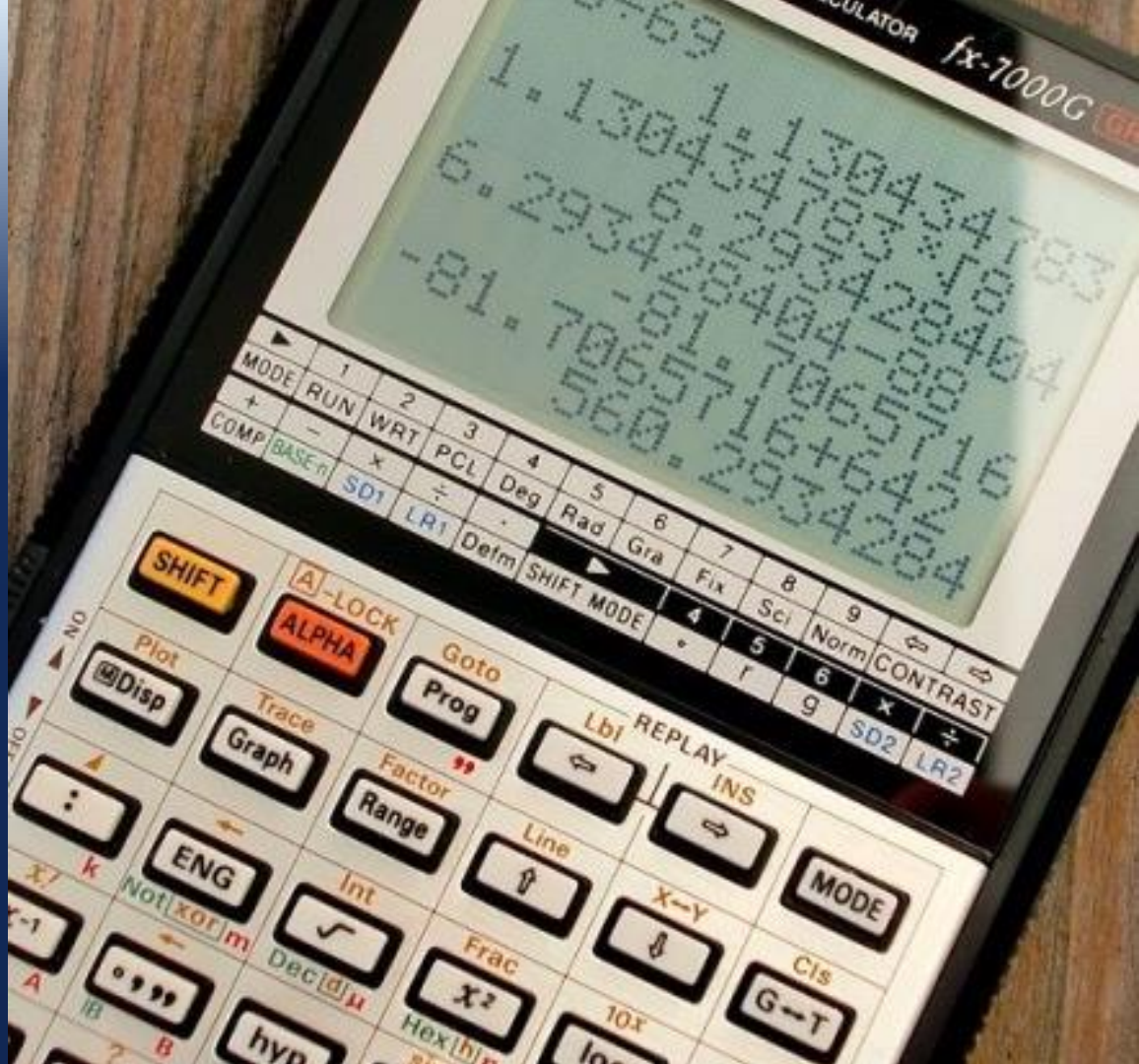
AIRCRAFT ENGINES

Projet de travail

Programmation scientifique

Jérôme Lacaille

SAFRAN



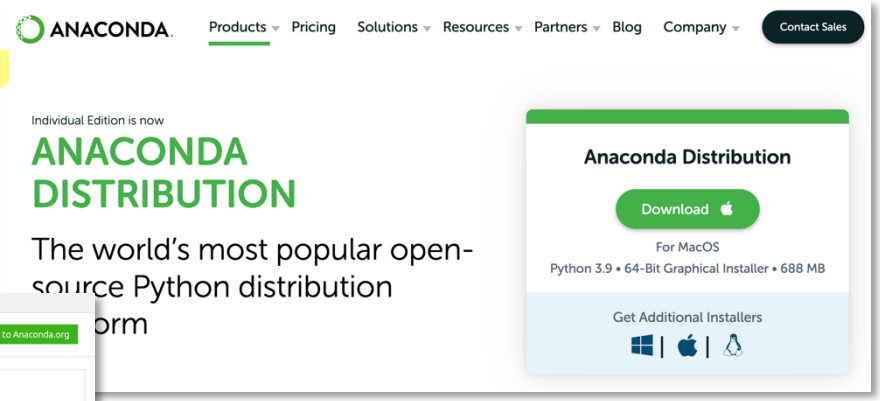
Visual Studio Code

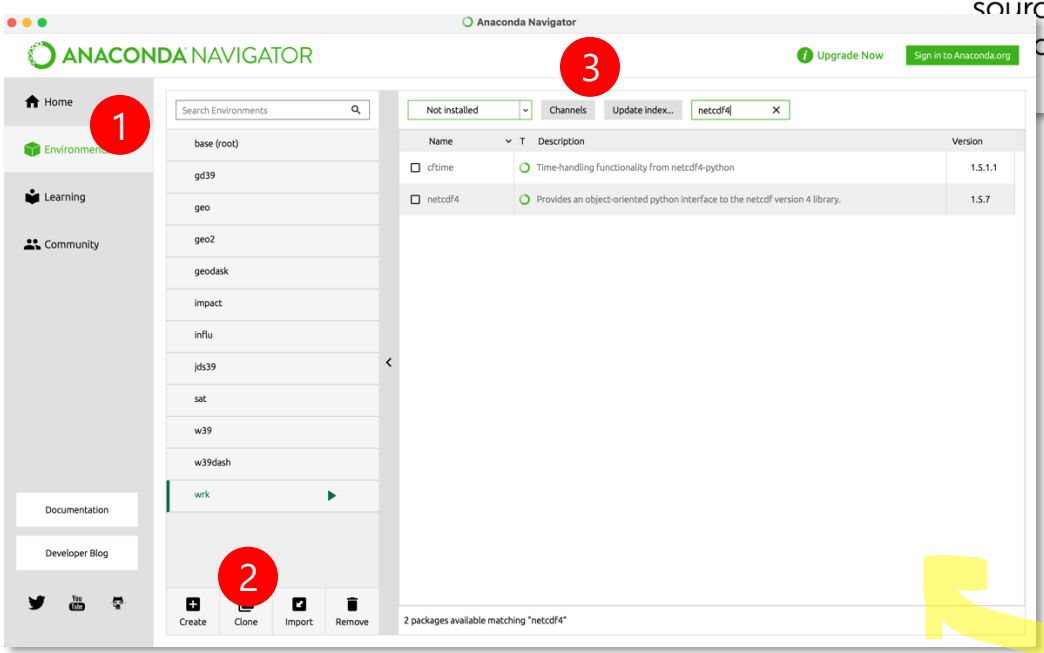
Chapitre 13

Installation d'un environnement Python

Installation de Python

- **Installer Anaconda**
  - https://www.anaconda.com/products/distribution





- **Créer un environnement de travail**
  - Sous Anaconda créez votre environnement avec les packages de base qui vous intéressent :
    - pandas, numpy, matplotlib, statsmodels, ipywidgets…

Préparation d'un répertoire de travail

Python permet de fabriquer des « packages ».

- Un package contient des « modules » de codes.
- Un package est un dossier contenant des codes (.py).
- Un tel dossier est un package s'il a un fichier « __init__.py » à la racine.

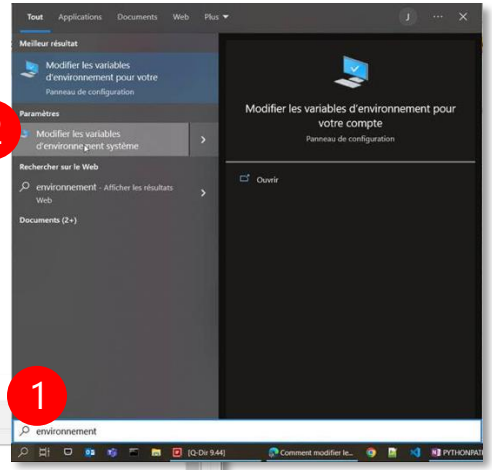
Les packages doivent être référencés.

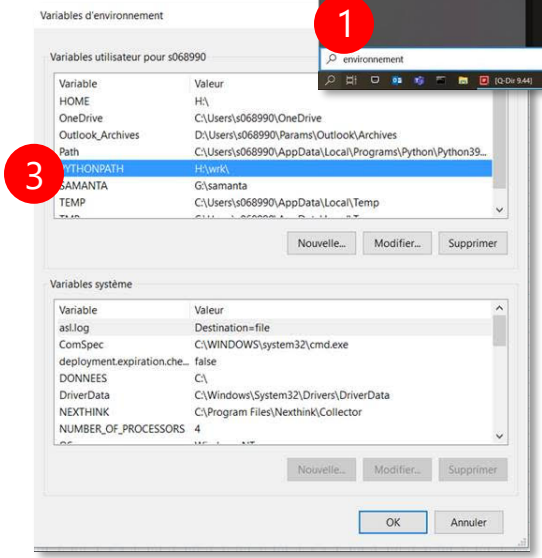
- Python recherche les package dans le dossier pointé par la variable d'environnement PYTHONPATH.
- Ensuite il ira chercher les packages téléchargés dans votre environnement.
- Donc si vous voulez développer un package il faut créer et instancier PYTHONPATH.

Création de la variable PYTHONPATH

- **Sous MacOS 11**
  - Lancer un Terminal
  - Créez le répertoire de travail
    - cd
    - mkdir WORK
  - Lancez un éditeur en ligne de commande
    - nano .bash_profile
      - (Vous pouvez utiliser vim sous Linux)
  - Modifiez le fichier « .bash_profile » en ajoutant la ligne
    - export PYTHONPATH = ~/WORK
  - Quittez l'éditeur par Ctrl-X et acceptez les modification (Y).

- **Sous Windows 10**
  - Dans la recherche, tapez environnement et sélectionnez





Il est toujours possible de placer le fichier de travail ailleurs.

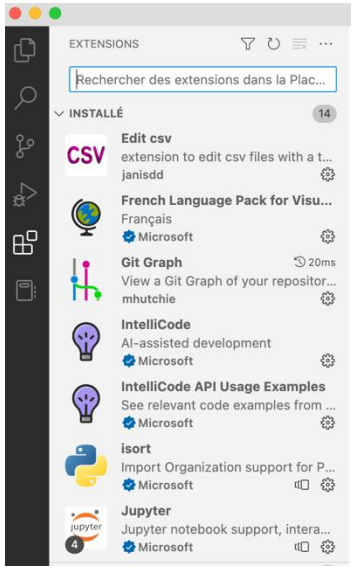
Création de la variable PYTHONPATH (suite)

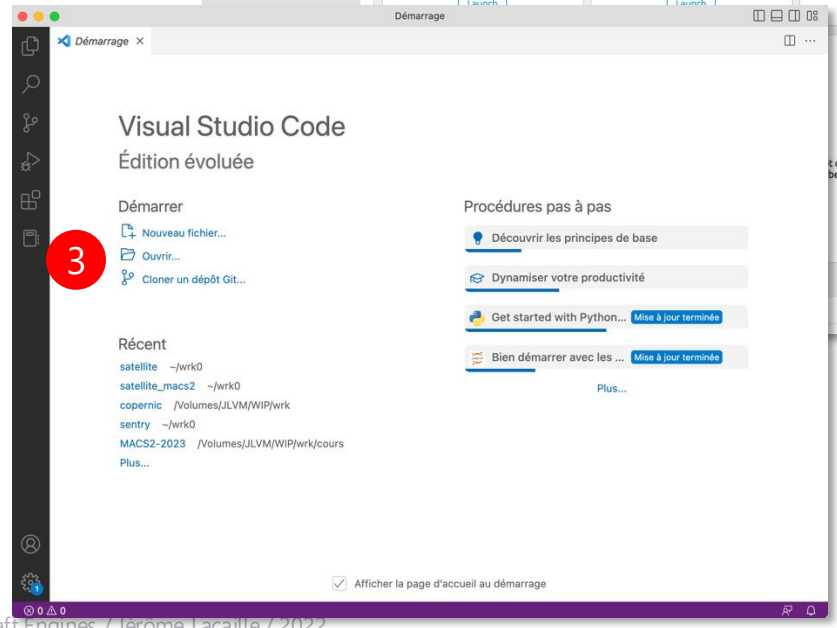
- **Avec le shell zsh**
  - zsh ets différent de bash et il est utilisé par dafaut sur les MacBooks les plus récents.
  - Le fichier à modifier est .zshrc
    - A créer éventuellement.
    - Il se trouve dans votre répertoire de base
      - Vous y allez par la commande « cd » dans un terminal.
      - Il est aussi référencé comme « ~ ».
  - La ligne à modifier est la même :
    - export PYTHONPATH=~/WORK
      - ~/WORK est le chemin dans lequel se trouve votre dossier « satellite ».

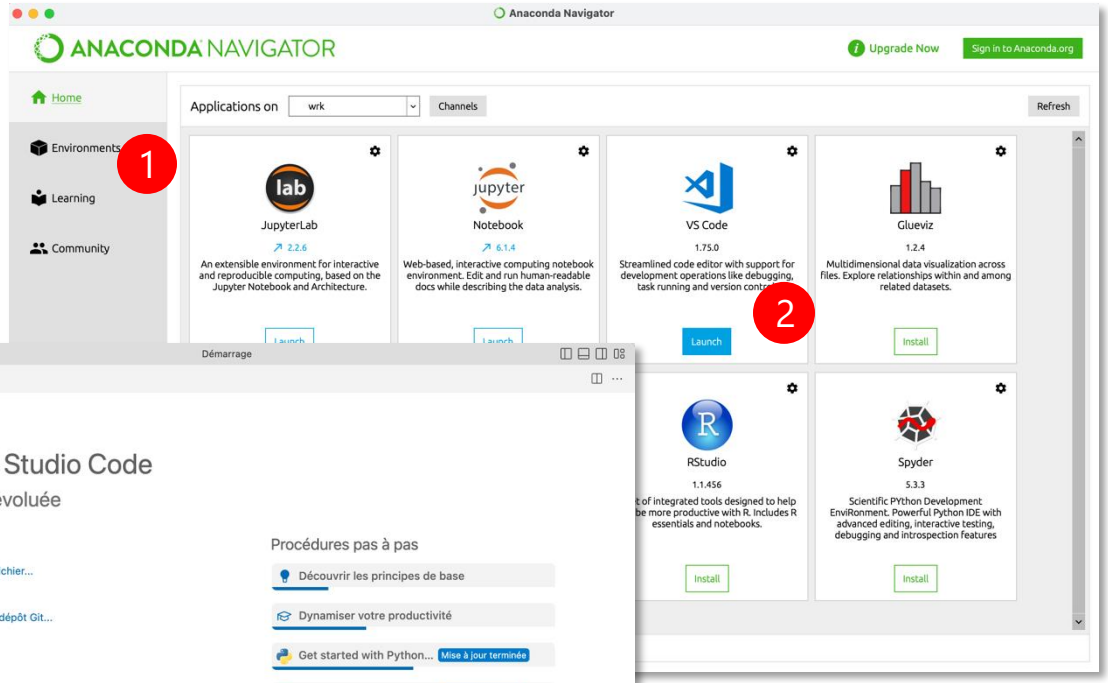
- **Sous Ubuntu**
  - Il faut rajouter une ligne dans le fichier
    - /etc/environnement
      - Le créer s'il n'existe pas.
      - Il faut les droits administrateur :
    - sudo vim /etc/environement
  - Rajouter la ligne
    - export PYTHONPATH=/User/…/WORK
      - /usr/…/WORK est le répertoire dans lequel se trouve votre dossier « satellite ».

Installation de VS-Code

- **Installez Visual Studio Code depuis Anaconda**
  - Installez ensuite les compléments que vous préférez.



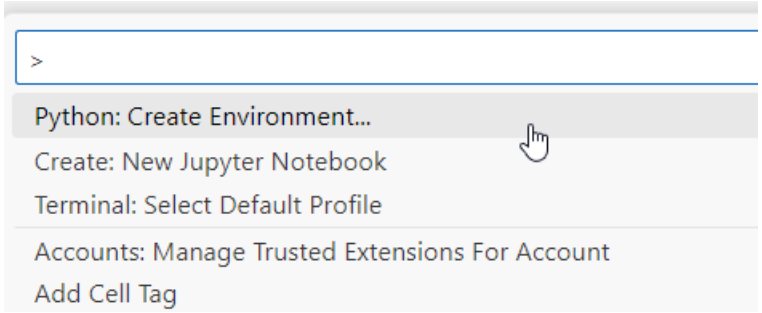


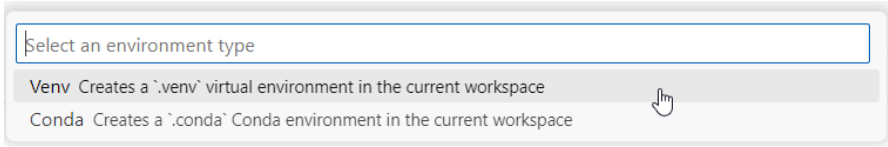


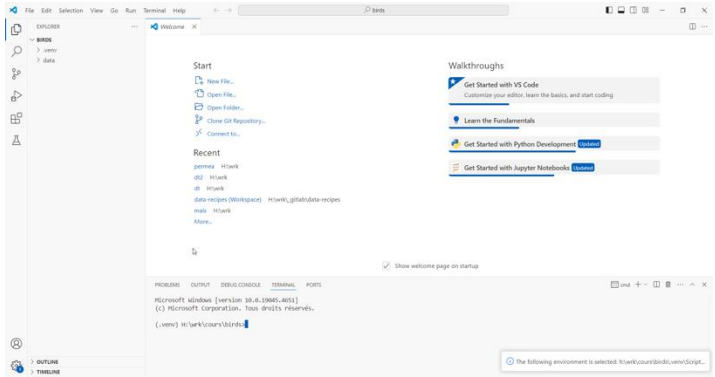
- **Ouvrez votre package**
  - Charger le répertoire pour commencer.

Création d'un environnement virtuel sous VSCode

- **Commencer par créer un répertoire de travail**
  - Ouvrez VSCode dans ce répertoire
- **Créer l'environnement:**
  - Shift-Ctrl-P → Python: Create Environment…
  - Sélectionner le type (pip ou conda)
- **Une fois dans votre nouvel environnement**
  - Ouvrez une console: Ctrl-ù
  - Installez vos packages.







Packages indispensables

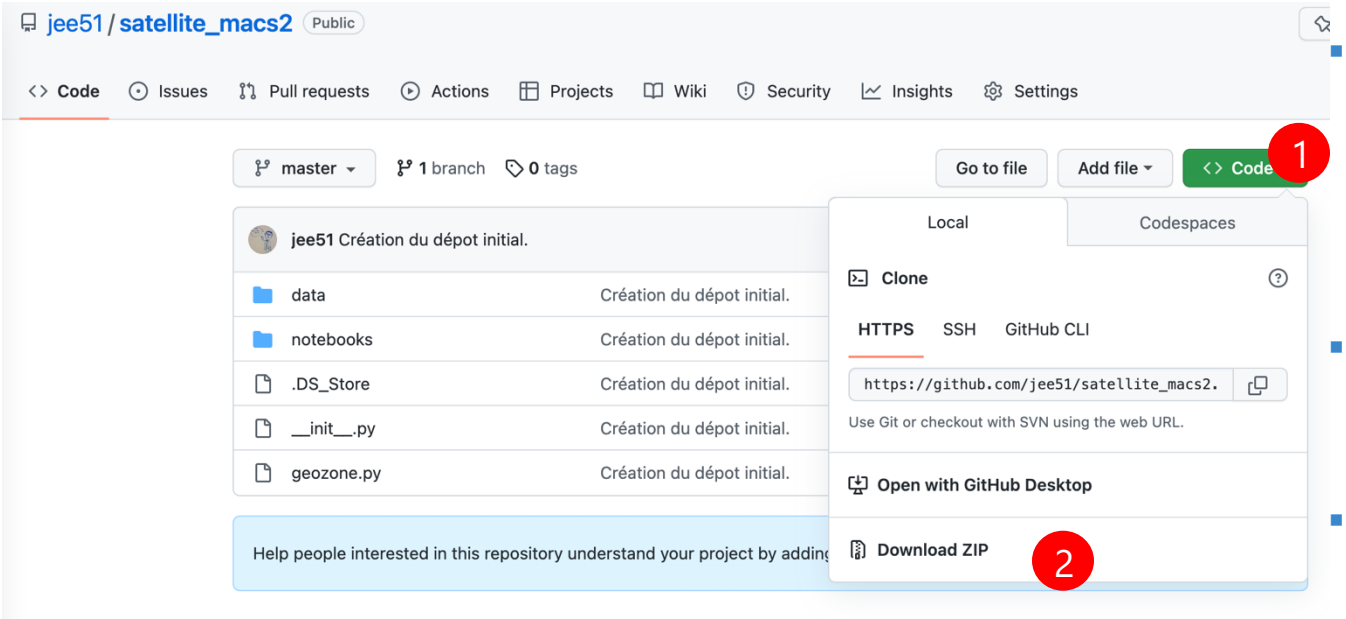
- **Packages de base**
  - pandas
- **Affichages**
  - matplotlib
  - plotly
- **Notebooks**
  - jupyter
- **Analyse de données**
  - scikit-learn
  - statsmodels
- **Scrapping**
  - beautifulsoup4
  - requests
  - lxml

Chapitre 14

Récupération d'un package python

Récupération du package

Pour vous aider dans le développement j'ai créé un package trivial nommé « satellite_macs2 » dans lequel on va pouvoir travailler.



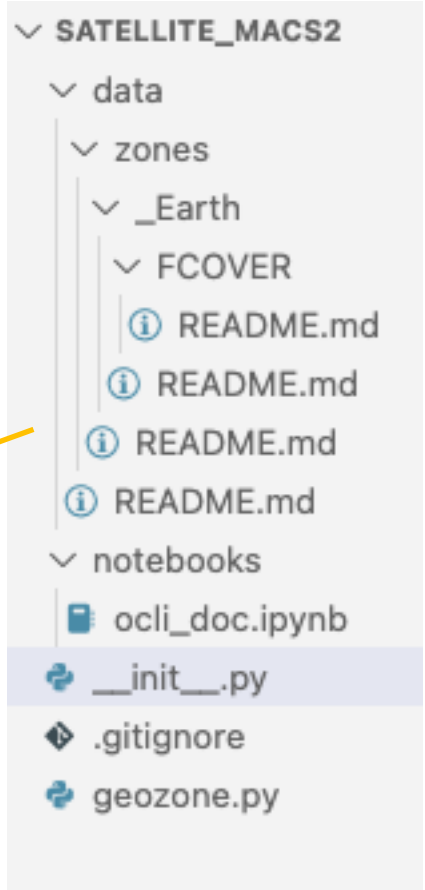
- Récupérer ce package sur GitHub
  - jee51/satellite_macs2 (github.com)

  Utilisez la fonction de téléchargement.
- Renommer le répertoire en « satellite »
  - Vous pourrez ensuite créer votre propre repository.
- Le répertoire de base doit se trouver dans votre PYTHONPATH.

Contenu de satellite_macs2

- **data**
  - Répertoire contenant les données.
    - zones (sous-répertoire pour les données géographiques)
      - _Earth : données globales (images et zones).
        - FCOVER : les images issues de GLS sur l'indicateur FCOVER.
- **notebooks**
  - Répertoire de travail
    - On y développe des notebooks de tests.

Notez la présence de fichiers README.md et leur effet dans GitHub



- **Dans le répertoire principal on place les fichiers du module.**
  - __init__.py déclare le sous-répertoire comme un nouveau package si celui-ci est dans un dossier pointé par la variable d'environnement PYTHONPATH.

__init__.py

__init__.py

```
1   from .geozone import EARTHDIR
```

- **Le fichier d'initialisation __init__.py sert à déclarer que le répertoire est désormais un nouveau package Python.**
- **Il peut être vide.**
- **S'il n'est pas vide son contenu est exécuté lors de la commande import du module.**
  - Dans notre cas si vous avez renommé ce répertoire en « satellite » il sera automatiquement exécuté quand vous ferez par exemple :
    - import satellite as sat
- **On se sert en général de ce fichier pour automatiser l'accès à des objets.**
  - « . » représente le répertoire courant.
  - « .geozone » correspond au module « geozone.py » du package « satellite »
    - La commande « from .geozone import EARTHDIR » sous-entend qu'un objet « EARTHDIR » existe dans le module « geozone » et que l'on pourra y accéder via un appel « sat.EARTHDIR » plutôt que « sat.geozone.EARTHDIR ».

.gitignore

- **Le fichier .gitignore est utilisé par git pour éviter de prendre en charge tous les éléments de votre répertoire.**
  - GitHub sait fabriquer automatiquement un « .gitignore » par défaut. C'est ce que j'ai fait, et j'ai rajouté quelques éléments perso à la fin pour notre projet.

.gitignore

```
117  # Rope project settings
118  .ropeproject
119
120  # mkdocs documentation
121  /site
122
123  # mypy
124  .mypy_cache/
125  .dmypy.json
126  dmypy.json
127
128  # Pyre type checker
129  .pyre/
130
131  # Perso
132  .DS_Store
133  *~
134  ._*
135  ~$*
136  SAFE/
137  ZIP/
138  FCOVER*OCLI*
```

geozone.py

- **Un premier module.**
  - Ce module va être développé en cours.
  - Il servira la gestion des zones géographiques captées depuis le site geojson.io (https://geojson.io/).

geozone.py > ...

```
1   # -*- coding: utf-8 -*-
2   """
3   GEOZONE - gestion des zones géographiques.
4
5   @author: Jérôme Lacaille
6   """
7
8   import os
9
10  EARTHREP = 'data/zones/_Earth/'
11  EARTHDIR = os.path.join(os.path.dirname(__file__),EARTHREP)
12
```

- **Bonnes pratiques**
  - En général on commence par un premier commentaire ligne précisant l'encodage.
  - Ensuite on met un champ texte décrivant le module.
    - Ce champ texte sera utilisé par les éditeurs pour l'aide en ligne.

- **Code**
  - Le code d'un module est exécuté à l'import.
    - Ce sera le cas ici quand on importera le package car le module »geozone » est importé dans « __init__.py ».
      - Vous retrouvez la variable EARTHDIR citée plus haut.

notebooks/geozone_doc.ipynb

- **Bonne pratique**
  - Je place un sous-dossier « notebooks » dans mes packages pour décrire le fonctionnement de mes modules.
    - En général j'utilise le nom du module testé avec un suffix « _doc ».
  - Ce premier notebook va nous aider à tester et créer le module « geozone » .

Test des fonctions d'accès aux données optiques.

Version 1.0 [31/01/2023] Jérôme Lacaille

[1]
```
import matplotlib.pyplot as plt

%matplotlib inline

plt.rcParams["figure.figsize"] = (9,6) # Pour l'affichage d'images un peu plus grandes.
```

[53]
```
# Ce bout de code est pratique car il permet de recharger automatiquement un package quand vous le développez.
%reload_ext autoreload
%autoreload 2
```

- **Imports**
  - Ici on importe juste « matplotlib » pour les affichages.
    - On ajoutera les autres packages nécessaires au fur et à mesure pour des raisons juste pédagogiques.
    - Notez la mise à jour de la taille des images à l'aide du paramètre « rcParams ».
  - La seconde commande magique (%) sert à recharger automatiquement les sous-modules du packages quand ils sont modifiés.
    - Regardez les commandes magiques (%magic).

Travaux dirigés

- **Ouvrez le fichier notebooks/geozone_doc.ipynb**
  - Suivez les indications.
  - Vous devrez progressivement ajouter des éléments au module « geozone.py » et au fichier d'initialisation « __init__.py ».
    - Attention, ce notebook a été réalisé avec un code fini donc si vous exécutez directement les cellules il y a de forte chance que cela ne marche pas du premier coup. Vous devrez coder d'abord. Les résultats vont donc disparaitre.
    - Je vous conseille de recopier les cellules avant de les exécuter pour garder une trace de la réponse attendue.
    - Vous pouvez aussi regarder le résultat directement dans GitHub.






Chapitre 15

Les objets Python

SAFRAN

Création d'une classe simple

```python
class MaClasse(ClasseDerivee,...):
    """ MaClasse - description.

        Un texte qui détaille la classe, ses propriétés et
        méthodes.
    """
    variable_de_classe = ...

    def __init__(self,parametre,...):
        """
            Initialisation de la classe.
        """
        # Propriétées.
        self.propriete_1 = ...

    # Méthodes internes.
    def __str__(self):
        """
            Génère un texte pour l'affichage de la classe.
        """
```

```python
def __getitem__(self,i):
    """
        Revoie le ième élément si la classe est une
        collection.
    """

def __iter__(self):
    """
        Démarre une itération et renvoie le premier
        élément.
    """

def __next__(self):
    """
        Poursuit l'itération et lève une exception
        StopIteration lors de la dernière.
    """

# Méthodes utilisateur.
def methode_1(self,...):
    """
        Définit une méthode spécifique.
    """
```

Création des erreurs

- **Les erreurs en Python génèrent des exceptions.**
  - De nombreuses catégories d'Exceptions existent et sont des classes qui dérivent toutes de la classe Exception.
    - ValueError, OSError, NameError, MemoryError, KeyError, AssertError…
- **Interception**
  - Pour traiter une erreur on utilise un bloc try/except/else/finally
    - try : teste l'instruction.
    - except : intercepte une exception (toutes par défaut)
    - else : si tout se passe bien
    - finally : exécuté à la fin dans tous les cas
- **Lancer une exception**
  - raise MonException(« message »)
  - assert instruction, « message en cas d'erreur. »

```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!")
    else:
        print("result is", result)
    finally:
        print("executing finally clause")
```

Chapitre 16

Git et GitHub

Commandes d'ajout et suppression

sha = le code du commit (hash code)

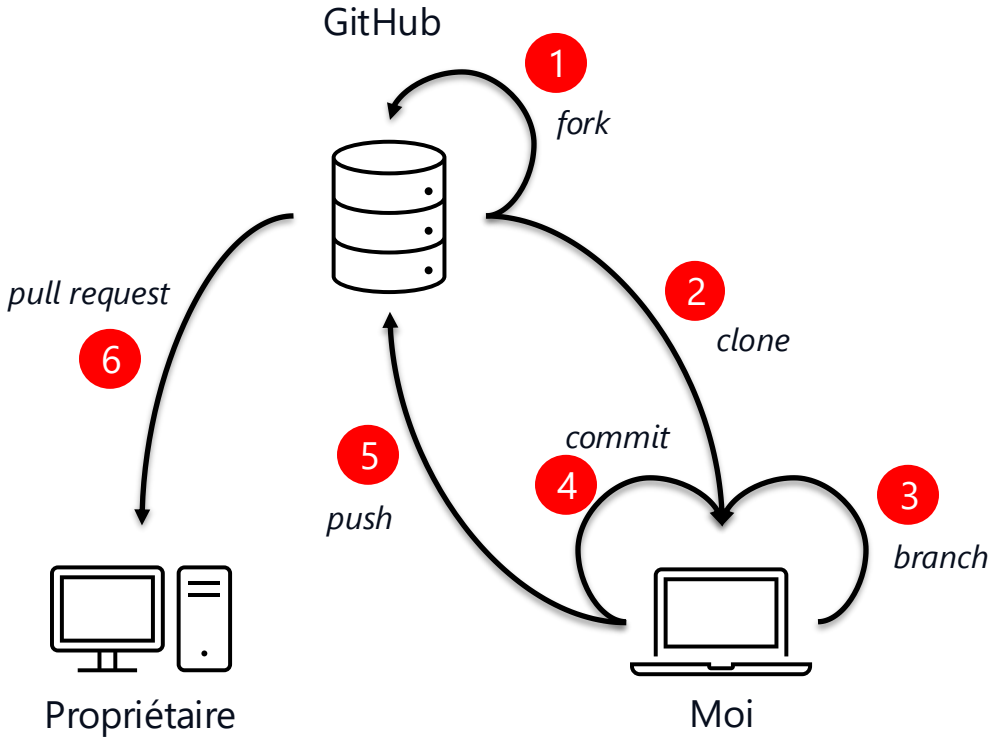
- **git status**
  - renvoie l'état du répertoire courant
- **git init**
  - initie un nouveau répertoire
- **git add**
  - ajoute un fichier au suivi
- **git commit –m « message »**
  - met à jour la version
    - git commit –am « … »
- **git log**
  - récupère les informations du répertoire
- **git remote add origin <url>**
  - connecte répertoires local et distant
- **git checkout <sha>**
  - revient à l'étape correspondant au sha donné
- **git checkout master**
  - retourne au dernier commit
- **git revert <sha>**
  - crée un nouveau commit qui revient au sha
- **git commit --amend –m « message »**
  - change le message du dernier commit
- **git reset --hard**
  - supprime toutes les modifications depuis le dernier commit

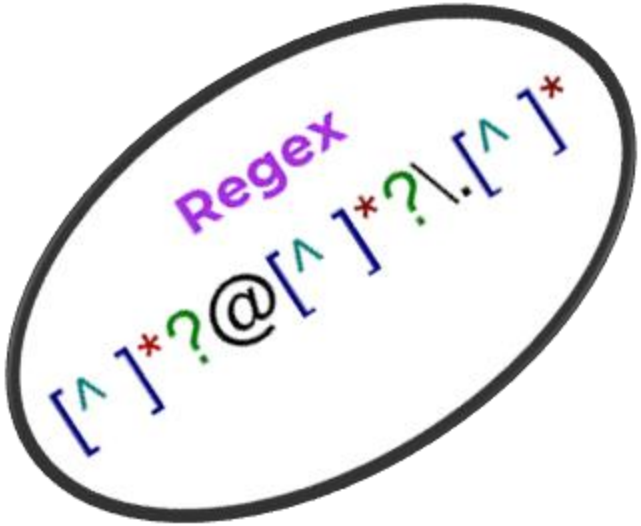
Connexions et branches

- **git clone <url>**
  - récupère un contenu distant
    - Pour créer un repository nouveau il est souvent plus facile de le créer sous GitHub et de le cloner.
- **git push origin master**
  - renvoie le code mis à jour (après commit) sur GitHub (master).
- **git pull origin master**
  - récupère le code qui a changé sur le master.
    - origin : est votre version locale en cours.
    - master : est le nom de la branche principale sur GitHub.
- **git blame un-fichier**
  - voir la liste des commits du fichier.
- **git show <sha>**
  - le détail du commit correspondant au sha
- **git branch**
  - renvoie la liste des branches
- **git branch ma-branche**
  - crée la nouvelle branche « ma-branche »
- **git checkout ma-branche**
  - passe en local sur la nouvelle branche
    - git checkout -b ma-branche
      - réalise les deux opérations précédentes en une seule commande
- **git checkout master**
  - on se place sur la branche principale
- **git merge ma-branche**
  - fusionne ma-branche dans master
    - Les conflits sont des balises >>>> à modifier.
- **git branch -d ma-branche**
  - supprime une branche
    - git branch -D ma-branche
      - pour forcer après un commit.

Synthèse

- **Sur GitHub**
  - fork : reproduit une version personnelle d'un répertoire
  - pull request : demande un tirage au propriétaire
- **Pour aider :**
  - git stash
    - sauvegarde les modifs en cours et revient au commit précédent
  - git stash pop
    - recharge les modifications sauvegardées sur la branche en cours.





Chapitre 17

Expressions régulières

SAFRAN

Expressions régulières

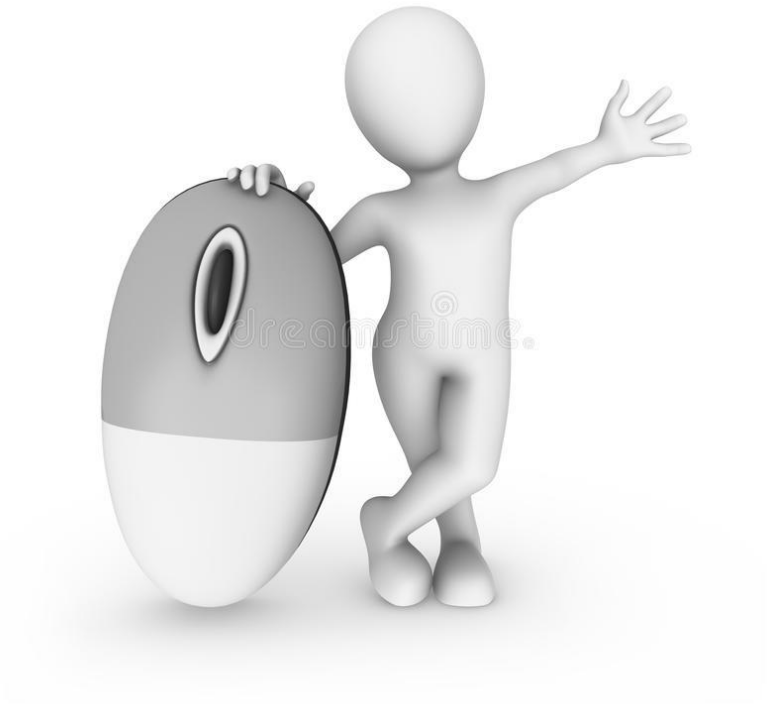
- **Les expressions régulières sont un outil puissant permettant d'extraire des parties de textes**
  - Dans notre cas les noms de fichiers sont formés ainsi :
    - FCOVER300_RT6_202102200000_GLOBE_OLCI_V1.1.1
  - On souhaite extraire la révision (RT6) donc 6 et la date (20/02/2021).
  - Une expression régulière utile est « (RT[0-9])_([0-9]+)_ »
    - m = re.search('(RT[0-9])_([0-9]+)_',fname)
  - On a deux groupes entre parenthèses.
    - Le troisième caractère du premier groupe est la révision.
    - Il faut ensuite convertir le second groupe
      - Par exemple avec
        - pd.to_datetime(m.group(2),format='%Y%m%d%H%M')

Expressions régulières (exemple)

```
1  # Pour extraire la révision et la date on recherche un pattern RT0_YYYYMMDD0*_ avec un
   chiffre à la place du 0.
2  print("Le nom du fichier :", filename)
3  m = re.search('(RT[0-9])_([0-9]+)_',filename)
4
5  print("Nombre de groupes :", m.lastindex)
6  print("La chaine entière :", m[0])
7  print("Le premier groupe :", m[1])
8  print("Le second groupe  :", m[2])
9  print("Tous les groupes  :", m.groups())
```

✓ 0.0s

```
Le nom du fichier : FCOVER300_RT6_202102200000_GLOBE_OLCI_V1.1.1
Nombre de groupes : 2
La chaine entière : RT6_202102200000_
Le premier groupe : RT6
Le second groupe  : 202102200000
Tous les groupes  : ('RT6', '202102200000')
```



Chapitre 18

Interactivité

Interactivité

- **Le package ipywidgets permet d'offrir une interactivité aux notebooks.**

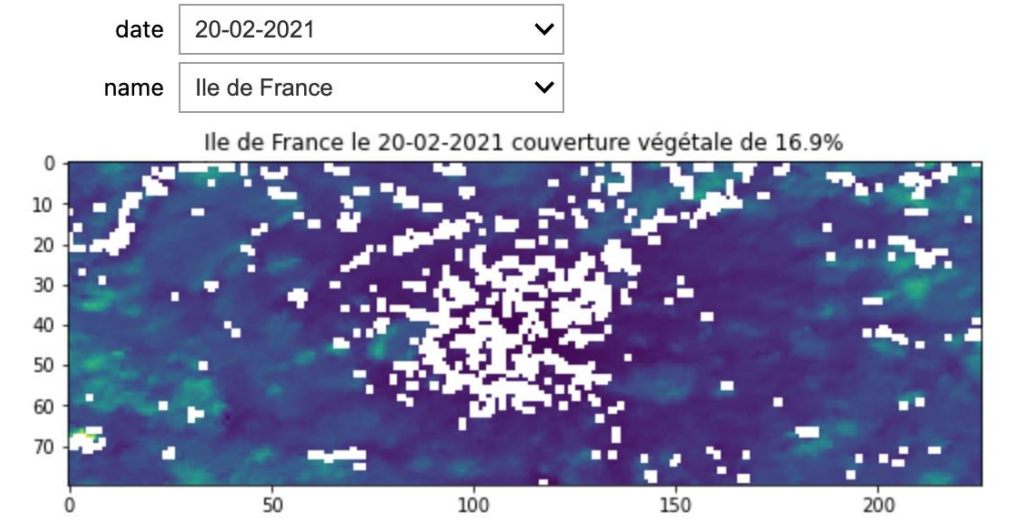
Exemple d'interactivités

- **Ocli**
  - Le notebook ocli_doc donne un exemple simple d'utilisation de l'interactivité.
- **Tabata**
  - Le package tabata qui est disponible en ligne sur gitHub vous permettra de créer vos propres animations.

```
import ipywidgets as widgets
```

```
1  widgets.interactive(nczplot, date=datelist, name=names)
```

✓ 0.5s



QUESTIONS ?

jerome.lacaille@safrangroup.com

SAFRAN