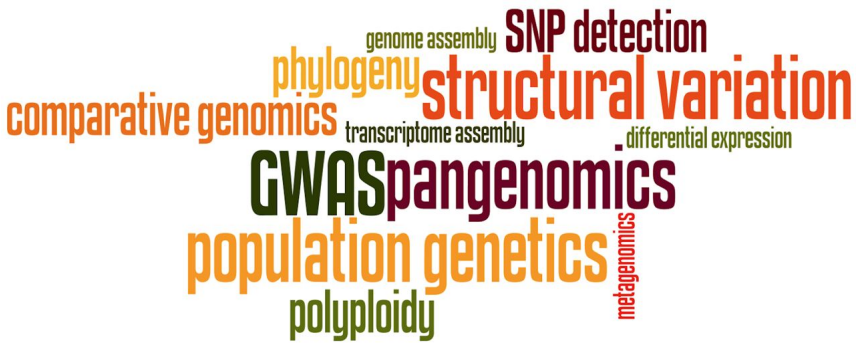
SouthGreen bioinformatics platform

# Session de formation 2019

# SouthGreen bioinformatics platform

**Bioinformatics platform dedicated to the genetics and genomics of tropical and Mediterranean plants and their pathogens**





Rice

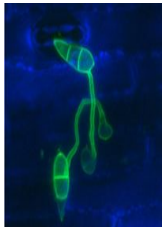

Banana


Palm


Sorghum


Coffee


*Cassava*


*Magnaporthe*

**www.southgreen.fr**

# SouthGreen bioinformatics platform

Charte

diade

Larmande Pierre
Sabot François
Tando Ndomassi
**Tranchant-Dubreuil Christine**

IPME

Comte Aurore
Dereeper Alexis

diade IPME

**Orjuela-Bouniol Julie**

agap

Bocs Stephanie
De Lamotte Frédéric
**Droc Gaetan**
Dufayard Jean-François
Hamelin Chantal
Martin Guillaume
Pitollat Bertrand
**Ruiz Manuel**
**Sarah Gautier**
Summo Marilyne

Bioversity International

**Rouard Mathieu**
Guignon Valentin
Catherine Breton

UMR BGPI

**Mahé Frédéric**
**Ravel Sébastien**

intertryp

Sempere Guilhem

Bioversity International

cirad

IRD Institut de Recherche pour le Développement FRANCE

INRA SCIENCE & IMPACT

# SouthGreen bioinformatics platform

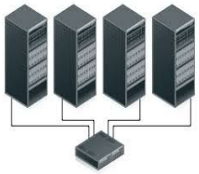
## Workflow manager

TOGGLE
Toolbox for generic NGS analyses

SNAKEMAKE

Galaxy

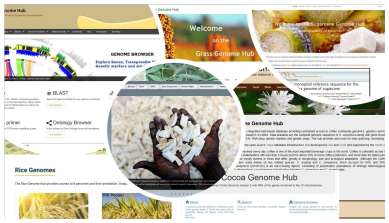
## HPC and trainings….



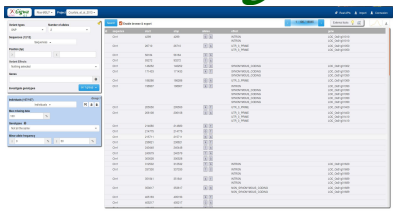

IRD Institut de Recherche pour le Développement

cirad

## Genome Hubs & Information System

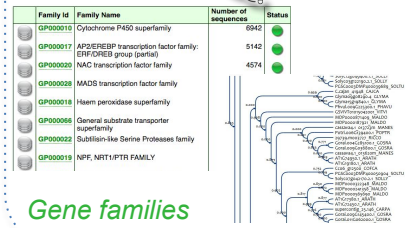



Gigwa



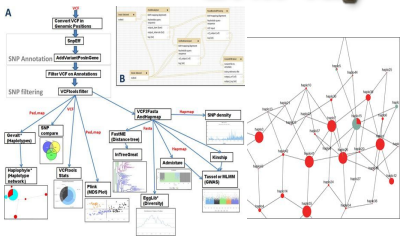
*SNPs and Indels*

GreenPhyl




*Gene families*

SNiPlay



https://github.com/SouthGreenPlatform

@green_bioinfo

***The South Green portal: a comprehensive resource for tropical and Mediterranean crop genomics***, Current Plant Biology, 2016

SouthGreen bioinformatics platform

ABiMS — Erwan Corre

IPME — Marie Simonin, Sébastien Cunnac

ASTRE Animal · Santé Territoires · Risques · Écosystèmes — Etienne Loire, Julie Reveillaud

UMR QualiSud — Florentin Constancias

diade — Valentin Klein

MiVEGEC Maladies Infectieuses et Vecteurs Écologie, Génétique, Évolution et Contrôle — Valérie Noël

TransVIHMI — Emmanuelle Beyne

i-Trop bioinformatics

**And more collaborators !**

SouthGreen bioinformatics platform

| | |
|---|---|
| 18-19/03 | Guide de survie à Linux - IRD |
| 21/03 | Initiation à l'utilisation du cluster CIRAD – CIRAD |
| 22/03 | Initiation à l'utilisation du cluster itrop - IRD |
| 15-16/04 | Initiation au gestionnaires de workflow SG & Gigwa – IRD |
| 18-19/04 | Guide du Jedi en Linux & bash - CIRAD |
| 13–16/05 | Python - IRD |
| 17/05 | Initiation aux analyses de données transcriptomiques – IRD |
| 21/05 | Utilisation avancée du cluster IRD – IRD |
| 23-24/05 | Initiation aux analyses de données métagénomiques – IRD |
| 6/06 | Manipulation de données et figures sous R – CIRAD |
| 25-27/09 | Assemblage et annotation de transcriptomes - IRD |

# Modules de formation 2019

- Toutes nos formations :
  https://southgreenplatform.github.io/trainings/
- Topo & TP : Linux For Dummies
- Environnement de travail : Logiciels à installer

Version 2019.3

# Formation Python

**Adapté du cours HLIN404 de Anne-Muriel Chiffoleau (UM)**

www.southgreen.fr

https://southgreenplatform.github.io/trainings

**Téléchargez les slides ici !** → https://southgreenplatform.github.io/trainings/python

# Pourquoi programmer en Python ?

Un des langages les plus utilisés, en bioinformatique et au delà

Qu'est-ce qu'un programme ou un script ?

- Une suite d'instruction que l'ordinateur doit exécuter
- => programmer = décrire à l'ordinateur très clairement tout ce qu'il doit faire
- Un programme est décrit par un algorithme (= suite d'instructions)

Pourquoi apprendre à programmer ?

- Faire travailler l'ordinateur à notre place (calculs)
- Automatiser / enchaîner des traitements (proposés par des outils existant)
- Effectuer des taches qu'aucun outils existant ne propose

Pourquoi en Python ?

- Un des langages le plus utilisés en bioinformatique car adéquat pour réaliser des scripts, portables sur Unix/Mac et sur Windows
- Syntaxe du langage incite à la clarté
- Langage interprété = programme traduit au fur et à mesure par un interpréteur afin d'être exécutée par l'ordinateur
- Langage orienté objet, moderne, haut niveau
- => Biopython : ensemble de fonctions et procédures conçues pour le traitement et l'analyse de données biologiques

Référence : http://python.developpez.com/tutoriels/cours-python-uni-paris7/

# Utiliser Python sur le cluster de l'IRD

- Se connecter au cluster
  Windows : utiliser PuTTY ou MobaXterm
  Linux : `ssh formationXX@bioinfo-master.ird.fr`

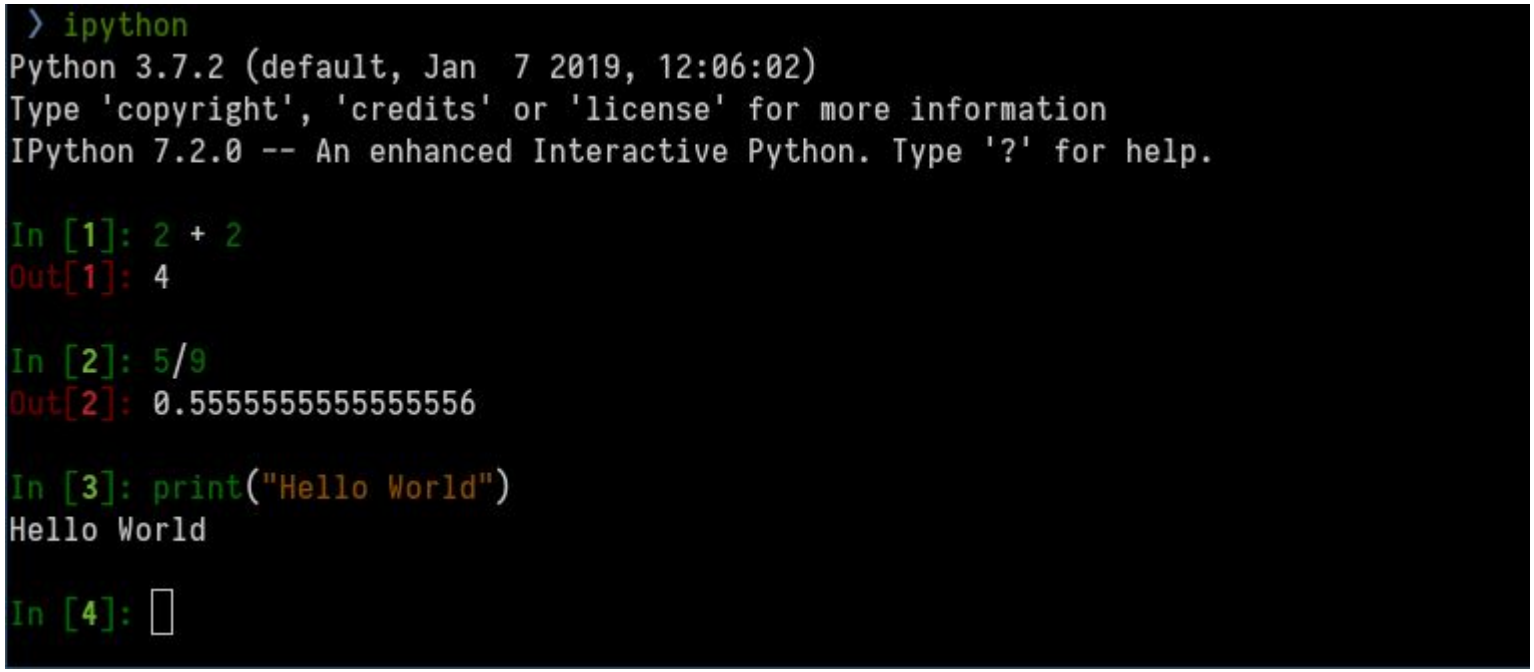
- Ouvrir une session interactive sur un noeud de calcul
  `qrsh -q formation.q`

- Charger le module de la bonne version de Python
  `module load system/python/3.6.5`

- Créer un dossier de travail
  `mkdir formation_python && cd formation_python`

# Premiers pas

- **Mode interactif** (REPL)
  Exécution d'instructions simples, retour en direct




Pour ce cours : utiliser `ipython` (interpréteur amélioré)

Sortir de l'interpréteur avec la commande quit() / exit() ou Ctrl+d

- **Mode programmation / script**
  exécution d'une suite d'instructions plus complexes écrites à l'avance

# Premiers pas

## Mode programmation / script

- Écrire les instructions dans un fichier `script_python.py` (`#` pour les commentaires)
- Exécuter le script :
  - Soit avec la commande : `python3 script_python.py`
  - Soit en rendant le fichier `script_python.py` « reconnaissable » (qui doit l'interpréter) et exécutable :
    - Ajouter en 1er ligne du fichier
      `#!/usr/bin/env python3`
    - Changer les droits du fichier
      `chmod +x script_python.py`
    - Lancer la commande : `./script_python.py`

# Utilisation de nano pour éditer des fichiers

nano : **éditeur de fichier** simple en ligne de commande.
Installé sur toutes les machines Linux.

```
nano unfichier.py
```

pour **éditer un fichier existant** ou **créer un nouveau fichier**

**Sauvegarder et quitter** : CTRL-X -> O -> Entrée

(les raccourcis sont affichés en bas de l'écran)

Chez vous, vous voudrez probablement utiliser un éditeur plus complet (vim, VS Code, Sublime Text…) ou un IDE (pyCharm, Spyder)

# Les variables

**Variable** : zone de mémoire dans laquelle on stocke une valeur
Une variable a un nom (ensemble de lettres, chiffres et _)

On déclare une variable en l'initialisant : `x = 2`
On peut changer sa valeur avec une affectation : `x = x + 3`

Python donne automatiquement un type a chaque variable en fonction de sa nature

- `w = 1` ⇒ type **int** (nombre entier) integer
- `x = 3.2` ⇒ type **float** (nombre flottant) float
- `y = "hello"` ⇒ type **str** (chaîne de caractères)
- `z = True` ⇒ type **bool** (booléen) boolean

# Les variables > Manipulation

Sur les nombres : opérateurs mathématiques de base

`+` `-` `*` `/` `**` (puissance) `%` (modulo)

Sur les chaînes de caractères :

`+` concaténation

`*` répétition

`type(ma_variable)` donne le type de la variable

Pas de conversion automatique ! (str + int ⇒ erreur)

Conversion : `int()`, `float()`, `str()`

# Les variables > Nommage

Un nom de variable ne peut pas commencer par un chiffre.
Éviter de commencer avec un _

Les noms de variables sont **sensibles à la case** (test ≠ TEST)

Utiliser la convention snake_case (sauf noms de classes)

```
ma_longue_variable = 3
```

Ne pas utiliser un **mot réservé** comme nom (`print`, `range`, `from`, `for`, …)

**Donnez des noms explicites à vos variables !**
`longueur_seq` plutôt que `l`

# Les fonctions

Les fonctions sont des **blocs de code isolés**, qui peuvent être **appelés** depuis n'importe où dans votre code. Une fonction prend des **arguments** en **entrée**, et peut renvoyer une **valeur de retour** en **sortie**.

Syntaxe :

```
x = fonction(argument1, argument2)
```

On peut appeler une fonction un nombre illimité de fois. Une fonction peut elle-même appeler d'autres fonctions.

# Un type spécifique : les classes/objets

Une **classe** est un "méta" type personnalisé que l'on peut définir via du code.

Une classe peut comporter :

- Des variables, appelées **attributs**
- Des fonctions, appelées **méthodes**

Une instance d'une classe est un **objet**

On peut appeler les attributs d'un objet en utilisant le . :

```
objet.attribut = 4
```

```
objet.méthode()
```

une variable `int` est en réalité une instance de la classe `int`.

**Tous les types en python sont des objets**

# Examiner les objets / Obtenir de l'aide sur ...

`help(obj)` : Affiche la documentation. Fonctionne sur les modules, les objets et les fonctions.
Raccourci sur IPython : `?obj`

`dir(obj)` : retourne la liste des attributs et méthodes de la classe de l'objet

# Entrées/sorties en console

Afficher du texte et des variables : `print()`

```
nom = "Valentin"
print("Bonjour", nom)
```

(print ajoute un espace entre les éléments automatiquement. Modifiable avec le paramètre "sep")

Récupérer une information saisie par l'utilisateur : `input()`

```
nom = input("Quel est votre nom ? ")
```

# Entrées/sorties > Affichage formaté

**Formatage** : Intégrer des variables dans une chaîne de caractères, en contrôlant la manière dont elles s'affichent

- **Fonction format()**

```
valeur = 34.67644
print("La valeur est {:.2f} !".format(valeur))
```

- **f-strings**

**! Python >= 3.6**

Même syntaxe que format, affiche les variables sans avoir à les passer en paramètre

```
valeur = 34.67644
print(f"La valeur est {valeur:.2f} !")
```

- *(opérateur % : ancienne syntaxe)*

# Pratique

Retrouvez les énoncés des exercices sur la page web de la formation :

**http://cpc.cx/oxd**

ou

https://southgreenplatform.github.io/trainings/python/pythonPractice/

# Pratique 1: le mode interactif

**En mode interactif, demander à l'interpréteur de calculer**

```
5*6
10/3
10.0/3.0
print("Hello world !")
```

Rappel mode interactif

ouvrir un terminal

lancer l'interpréteur python via la commande ipython

taper les instructions souhaitées

# Pratique 1: premier script

## Exercice 1

**Créer un programme python qui affiche "Hello world"**

Créer un fichier **hello.py** avec votre éditeur de texte (nano, …)

Taper l'instruction `print("Hello world")` dans le fichier

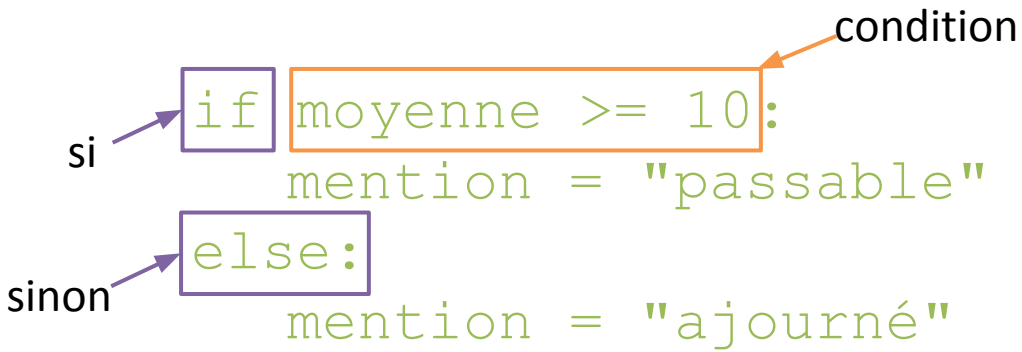
Enregistrer le fichier

Dans le terminal, se déplacer dans le répertoire où se trouve mon script (commande cd …)

Exécuter le script en tapant la commande **`python3 hello.py`**

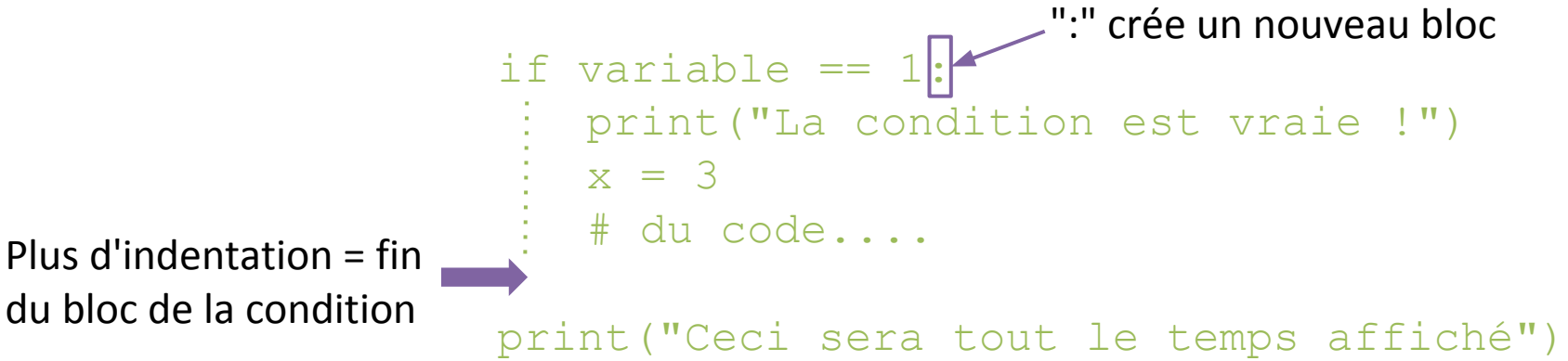
# Structures conditionnelles

Permet d'exécuter du code différent en fonction du résultat d'une ou plusieurs conditions/tests



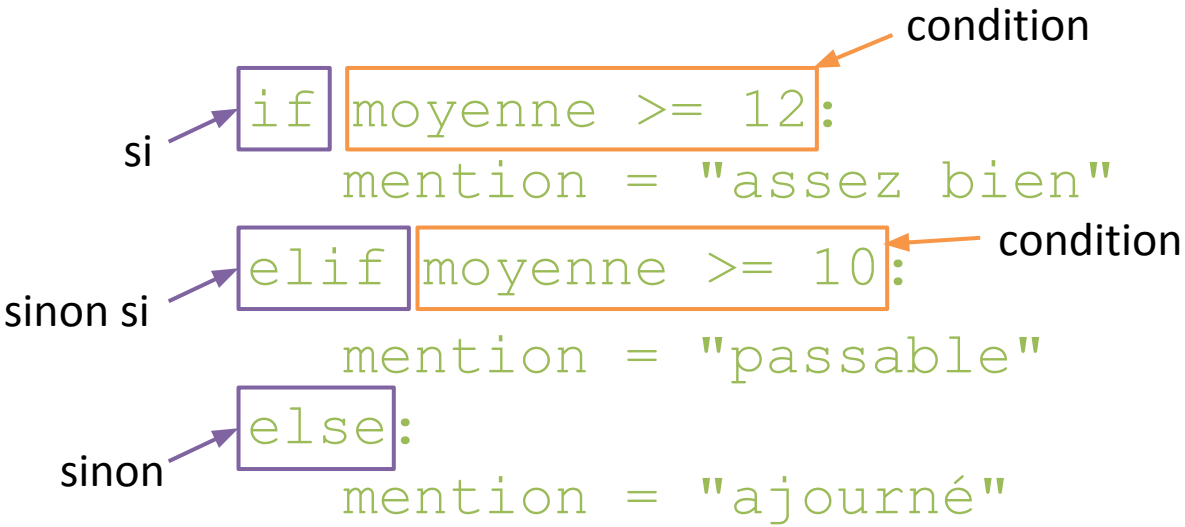
# Indentation en Python

Contrairement à la plupart des autres langages,
les **Blocs de code** (conditions, boucles, fonctions…) définis par l'**indentation**




Convention : Indenter avec **4 espaces** (pas de tabulations)

# Structures conditionnelles



## Opérateurs de comparaison

| | |
|---|---|
| `==` | égal à |
| `!=` | différent de |
| `>` | strictement supérieur à |
| `>=` | supérieur ou égal à |
| `<` | strictement inférieur à |
| `<=` | inférieur ou égal à |

**Attention !**

Ne pas confondre :

`=` opérateur d'assignation

et

`==` opérateur de comparaison

`is` est le même objet que (utiliser pour vérification de type)

```
x = 3
if type(x) is int:
```

# Structures conditionnelles > Opérateurs logiques

Les opérateurs de comparaisons renvoient un **booléen** (bool)

On peut combiner plusieurs conditions avec les **opérateurs logiques**

`and` ET
`or` OU (inclusif)
`not` NON


```
age = 32
if age >= 4 and age <= 99:
   print("Vous pouvez jouer aux legos")
```

| | |
|---|---|
| `True and True` | `True` |
| `True and False` | `False` |
| `False and True` | `False` |
| `False and False` | `False` |

| | |
|---|---|
| `True or True` | `True` |
| `True or False` | `True` |
| `False or True` | `True` |
| `False or False` | `False` |

| | |
|---|---|
| `not True` | `False` |
| `not False` | `True` |

# Pratique 2: les tests

## Exercice 2

**Créer un programme python exo2.py qui affiche la moyenne de 3 notes données par l'utilisateur.**

Voici le code python3 pour demander 3 valeurs à l'utilisateur qui seront stockées dans les variables note1, note2, note3

```
note1 = input("Donner une note : ")
note2 = input("Donner une note : ")
note3 = input("Donner une note : ")
```

Attention les variables note1, note2, note3 sont de type **chaîne de caractère** (str)

Modifier ensuite le programme afin qu'il affiche

"ajourné" si la moyenne est inférieure à 10
"passable" si la moyenne est supérieure ou égale à 10
"assez bien" si la moyenne est supérieure ou égale à 12
"bien" si la moyenne est supérieure ou égale à 14
"très bien" si la moyenne est supérieure ou égale à 16

# Structures de données: Listes

**Liste** : structure de données contenant une série de valeurs ordonnées (possiblement de types différents)

```
ma_liste = [1, "deux", 3.0]
ma_liste_vide = []
```

Appeler les éléments d'une liste

`ma_liste[0]` premier élément de la liste
`ma_liste[1]` second élément de la liste

Attention : Les index commencent à 0 !

# Listes

i,j = positions dans la liste x = valeur

Récupérer la longueur d'une liste :

```
len(ma_liste)
```

Prendre une tranche d'une liste :

`ma_liste[i:j]` ⇒ éléments de l'indice i à l'indice j-1

`ma_liste[i:]` ⇒ tous les éléments à partir de l'indice i

`ma_liste[:j]` ⇒ tous les éléments jusqu'à l'indice j-1

`ma_liste[:-1]` ⇒ tous les éléments sauf le dernier

`ma_liste[-1]` ⇒ dernier élément

Concaténer des listes

```
liste_1 + liste_2
```

# Listes

i = position dans la liste  x = valeur

Ajouter un élément à la fin d'une liste

```
ma_liste.append(x)
```

Remplacer la valeur d'un élément à une position déterminée

```
ma_liste[i] = x
```

Insérer un élément à une position déterminée

```
ma_liste.insert(i, x)
```

# Listes

i = position dans la liste  x = valeur

Trier une liste en place

```
ma_liste.sort()
```

Créer une copie triée d'une liste

```
liste_triee = sorted(ma_liste)
```

Inverser une liste

```
ma_liste.reverse()
```

Supprimer un élément d'une liste

```
del ma_liste[i]
ma_liste.remove(x)
```

(ne retire que la première occurrence de x)

# Listes

i = position dans la liste  x = valeur

Tester si un élément appartient ou pas à une liste

```
if x in ma_liste:
if x not in ma_liste:
```

Compter le nombre d'occurrences de x dans une liste

```
ma_liste.count(x)
```

**Attention** : toutes les recherches par valeur dans la liste sont en O(n)
= de **plus en plus lentes** au fur et à mesure que la taille de la liste augmente

# Listes

Une chaîne de caractères est une **liste non modifiable**

```
sequence = "ATGGCAT"
sequence[2]                    => "G"
```

On peut créer une liste modifiable à partir de la chaîne

```
liste_sequence = list(sequence)
```

## Listes de listes

```
liste = [[1, 2, 3], [4, 5, 6]]
liste[0]              => [1, 2, 3]
liste[0][0]           => 1
liste[1][2]           => 6
```

# Boucles

On peut avoir besoin de répéter une action (un bloc de code) plusieurs fois

Exemples:

- Afficher chaque élément d'une liste
- Tant que vous n'aurez pas compris les boucles, je rééexpliquerai

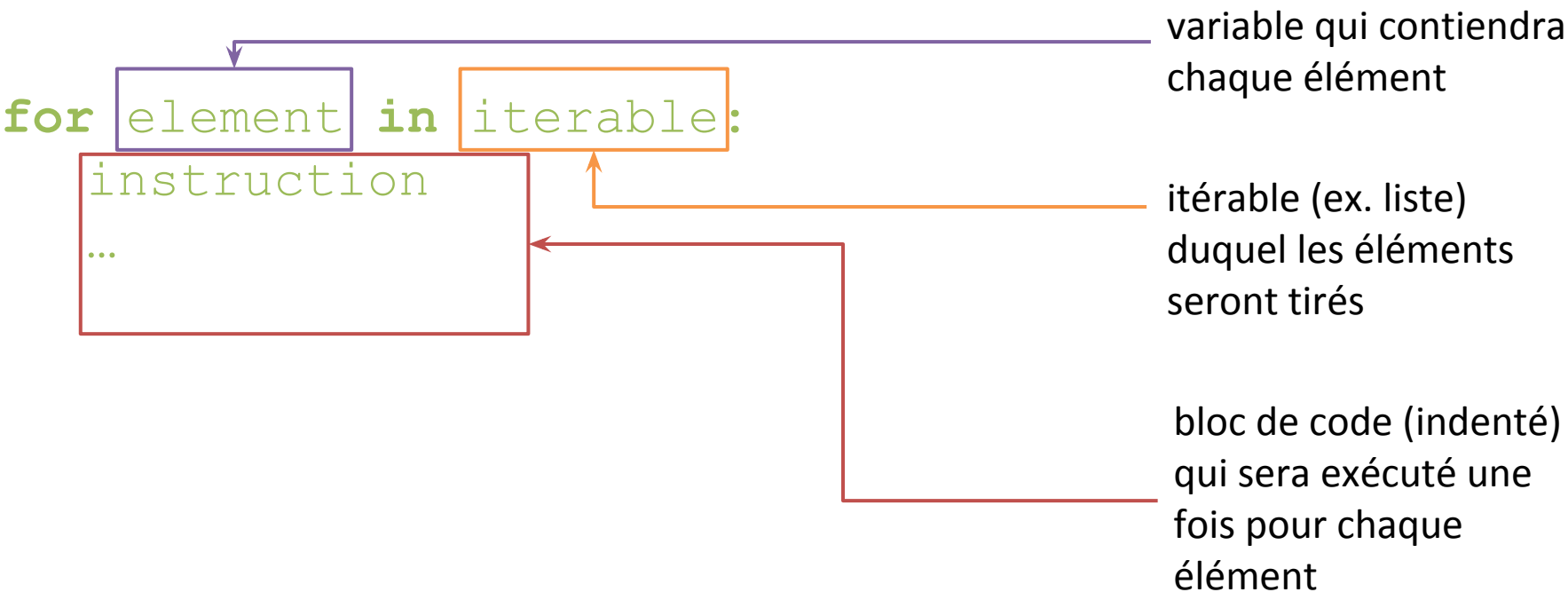
Pour cela on va "**boucler**".

Deux structure itératives : **Pour** (for) et **Tant que** (while)

# Boucles > for (POUR)

**Pour chaque élément** d'un itérable, exécuter un bloc de code

(une liste est un exemple d'itérable)




```
animaux = ["Lion", "Chèvre", "Vache"]
for animal in animaux:
    print(animal)
```

# Boucles > for (POUR)

Astuce : avoir en même temps la position et la valeur de l'élément

```
for position, element in enumerate(iterable):
    print(f"Position {position} : {element}")
```

**Attention :** Ne pas modifier la liste sur laquelle on boucle !

# Boucles > for (POUR)

Si on veut itérer sur un nombre croissant, on peut utiliser `range` :

`range(fin)`

Entiers de 0 à fin - 1

`range(début, fin)`

Entiers de début à fin - 1

`range(début, fin, pas)`

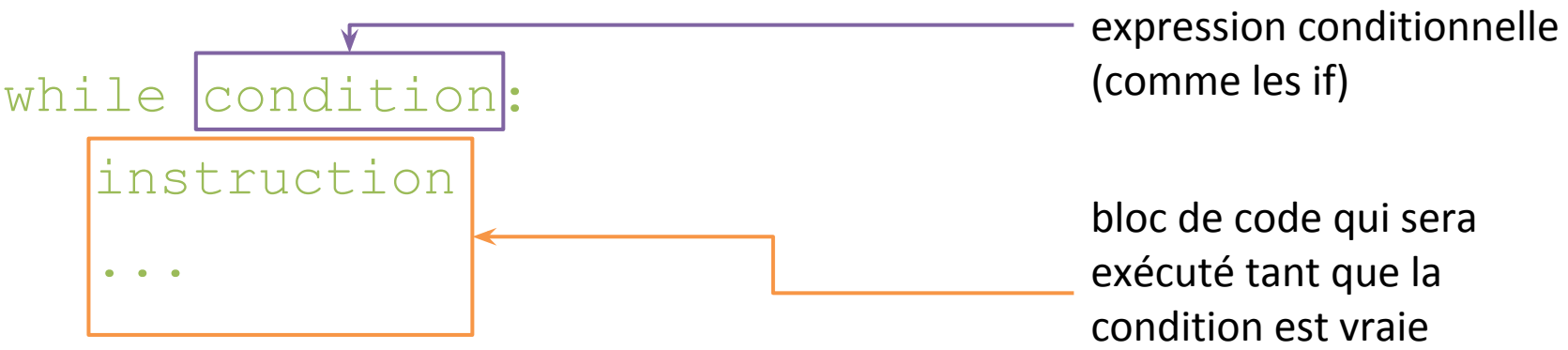
Entiers de début à fin - 1 tous les pas

Exemple : nombres pairs entre 2 et 12

```
for i in range(2, 14, 2):
    print(i)
```

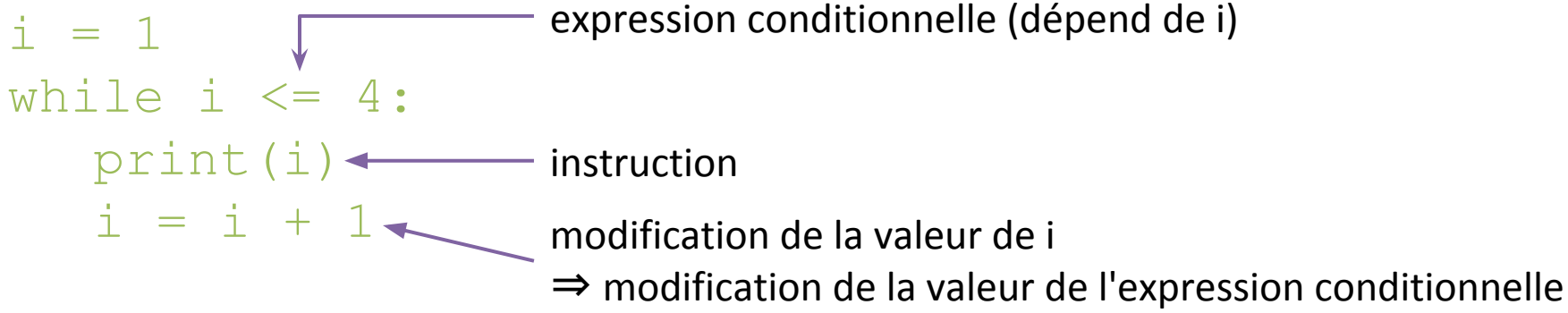
# Boucles > while (TANT QUE)

**Tant que** la condition est vraie, le bloc de code est exécuté




Le bloc d'instructions doit **modifier la valeur de l'expression conditionnelle**, sinon boucle infinie !

(rappel : **CTRL+C** pour arrêter un programme bloqué dans une boucle infinie)



# Boucles > Interruptions

Deux instructions qui permettent **d'interrompre une boucle prématurément** :

`continue` arrête l'exécution de l'itération en cours et **passe à l'itération suivante**

=> utile si on veut ignorer une variable particulière et passer directement à la suivante

`break` arrête l'exécution de l'itération en cours et **sort de la boucle**

=> utile par exemple si on cherche quelque chose et qu'on l'a trouvé

# Boucles > Interruptions

```
ma_liste = ["machin", "truc", "chose"]

for element in ma_liste:
    if element == "truc":
        continue
    print(element)

print("fini")
```

# Boucles > Interruptions

```
ma_liste = ["machin", "truc", "chose"]

for element in ma_liste:
    if element == "truc":
        print(element)
        break

print("fini")
```

# Pratique 3: les listes et les boucles

## Exercices 3 & 4

Soit la liste suivante

```
liste_animaux = ['vache','souris','levure','bacterie']
```

Créer un programme python *exo3_for.py* qui affiche l'ensemble des éléments de la liste liste_animaux en utilisant la boucle **for**

Créer un programme python *exo3_while.py* qui affiche l'ensemble des éléments de liste liste_animaux en utilisant la boucle **while**

Soit la liste de nombres suivante

```
impairs = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21]
```

Créer un programme python *exo4_pair.py* qui, à partir de la liste impairs, construit une nouvelle liste pairs dans laquelle tous les éléments de impairs sont **incrémentés de 1**. Une fois la nouvelle liste créée, l'afficher.

# Pratique 4: les listes et les boucles suite

## Exercices 5, 6 & 7

- Créer un programme python *exo5.py* qui à partir de la séquence nucléique `"TCTGTTAACCATCCACTTCG"` (chaîne de caractères) crée sa séquence complémentaire inverse, puis l'affiche.

- Créer un programme python *exo6.py* qui compte le nombre de chacune des bases (A, T, G, C) présente dans la séquence nucléique `"TCTGTTAACCATCCACTTCG"`, puis qui affiche ces nombres. Vous ferez une version qui utilise la fonction **count()** des listes et une autre version qui ne l'utilise pas.

- Créer un programme python *exo7.py* qui vérifie si la sous-séquence "ATG" (codon start) est présente dans la séquence nucléique `"TCTGTTATGACCATCCACTTCG"`, puis affiche "oui" ou "non" en fonction du résultat. Par la suite, modifier le programme pour qu'il affiche également la position de la sous-séquence "ATG" si elle est présente.

# Pratique 4: les listes et les boucles suite

## Exercice 7bis

- Reprendre l'un des programmes e*xo5.py, exo6.py* ou *exo7.py* et faire en sorte qu'il fonctionne pour plusieurs séquences (les séquences seront stockées dans une liste et pour chacune des séquences sera analysée)

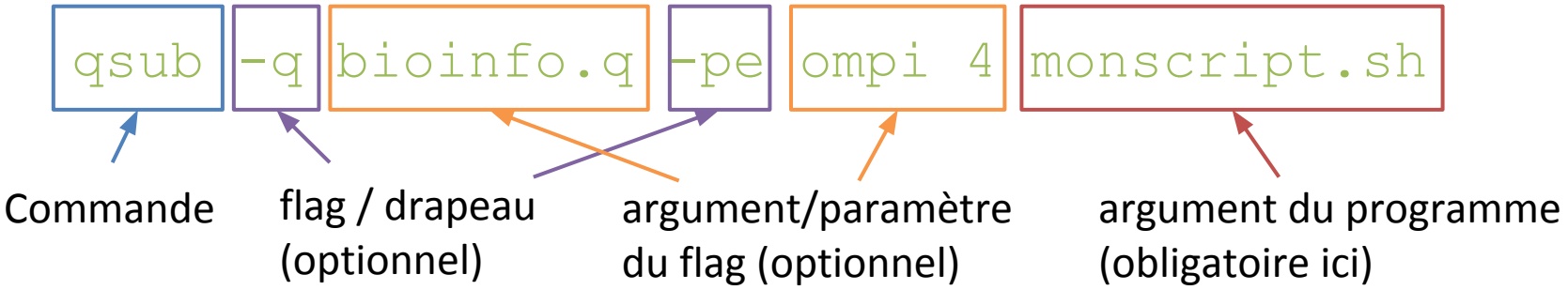
  Exemple :

```
seq1= "ATGCGTAGTCGT"
seq2= "AGGTTCGTATG"
mesSeq = [seq1,seq2]
```

# Passage d'argument

Jusqu'à présent, on utilise des données fixes ou demandées à l'utilisateur avec input()

On veut passer des **arguments/paramètres** au programme




- pour permettre l'exécution du programme avec **différentes données** sans avoir à modifier le script
- pour **éviter l'interaction avec l'utilisateur**

# Passage d'arguments

Exemples

```
note1 = input("Donner une note : ")
note2 = input("Donner une note : ")
note3 = input("Donner une note : ")
moy = (int(note1)+int(note3)+int(note2))/3
print ("La moyenne est ",moy)
```

On souhaite passer ces valeurs en argument au programme

# Passage d'arguments

Exemples

```
liste_animaux = ['vache','souris','levure','bacterie']
for nom in listeAnimaux:
    print(nom)
```

On souhaite passer ces valeurs en argument au programme

# Modules Python

Python met à disposition des **modules** qui contiennent des **fonctions souvent utilisées** ou pour **interagir avec le système**
(modules = bibliothèques/librairies)

Utilisation d'un module :

- **Installer** le module si il n'est pas dans la librairie standard (`pip install`)
- **Importer** le module en début de programme
  `import nom_module`
- **Appeler** un objet/une fonction d'un module
  `nom_module.fonction()`
- **Aide** sur un module
  `help(nom_module)`
  sur IPython : `?nom_module`

# Modules Python

Quelques modules de la librairie standard utiles :

- `math` : fonctions et constantes mathématiques de base (sin, cos, exp, pi…)
- `random` : génération de l'aléatoire (nombres, choix…)
- `sys` : passage d'argument, gestion de l'entrée/sortie standard…
- `os` : dialogue avec le système d'exploitation (système de fichiers, variables d'environnement…)
- `subprocess` : lancer d'autre programmes à partir de Python
- `re` : utilisation des expressions régulières
- `csv`, `json` : lecture et écriture des formats éponymes

Tous les modules de la librairie standard et leur documentation :

https://docs.python.org/3/library/index.html

# Passage d'arguments

Récupérer les valeurs passées en argument au programme :
Utilisation de la liste `sys.argv` du module sys (`import sys`)

- `sys.argv[0]` ⇒ nom du programme
- `sys.argv[1]...sys.argv[n]` ⇒ valeurs des arguments

Exemple pour la commande `python exo3.py 15 17 8`

- `sys.argv[0] = "exo3.py"`
- `sys.argv[1] = "15"`
- `sys.argv[2] = "17"`
- `sys.argv[3] = "8"`

# Passage d'arguments

Vérifier qu'on a le **bon nombre d'arguments**
= taille de `sys.argv`

Si on a 3 arguments, alors `len(sys.argv) == 4`

Si on a pas le bon nombre, quitter le programme :

`sys.exit()`
sortie normale

`sys.exit("Message d'erreur")`
sortie avec erreur (met le code d'erreur du shell à 1)

# Pratique 5: le passage de paramètre

## Exercice 5bis

- Reprendre le programme *exo5.py* et créer un nouveau programme python *exo5_arg.py* qui permettra de donner la séquence nucléique étudiée en argument du programme.

- Modifier le programme précédent *exo5_arg.py* afin de vérifier la présence du bon nombre d'arguments

# Pratique 5: le passage de paramètre

## Exercice 8

- Créer un programme python *exo8.py* qui permet de comparer 2 séquences données en argument et qui affiche le nombre de résidus identiques à la même position entre les 2 séquences (on considère que les 2 séquences sont de même longueur)

  Exemple :

  - avec les séquences `"ATTGCTA"` et `"ATTGCTC"`, le programme doit afficher "6 résidus identiques"
  - avec les séquences `"TAATTGT"` et `"ATTGCTC"`, le programme doit afficher "0 résidus identiques"

- Modifier le programme précédent *exo8.py* afin d'afficher le pourcentage d'identité (nb résidu identique / nb de résidu total)

- Modifier le programme précédent *exo8.py* afin d'autoriser des séquences de longueurs différentes

# Fichiers

Souvent en bioinfo, on doit **lire ou écrire des fichiers**

Ouvrir un fichier :

```
fd = open("nom_du_fichier", "mode d'ouverture")
```

Modes d'ouverture :

- `"r"` pour **lecture**
- `"w"` pour **écriture**, **écrase** le fichier si il existe déjà
- `"a"` pour **écriture**, **ajoute** à la fin du fichier si il existe déjà
- `"x"` pour **écriture**, **échoue** si le fichier existe déjà

Python traite les fichiers comme des fichiers texte par défaut (encodage utf-8)

# Fichiers

Souvent en bioinfo, on doit **lire ou écrire des fichiers**

Ouvrir un fichier :

```
fd = open("nom_du_fichier", "mode d'ouverture")
```

`fd` est un objet représentant le fichier ouvert sur lequel on peut appeler des fonctions pour manipuler le fichier

Une fois les manipulations finies, on doit impérativement **fermer le fichier** !

```
fd = open("fichier.txt", "r")
# ici instructions utilisant le fichier…
fd.close()
# ici on ne peut plus utiliser fd
```

## Lecture du fichier (fichier ouvert en mode `"r"`)

Le fichier a un curseur qui avance à chaque opération de lecture.

Pour revenir au début : `fd.seek(0)`

Lire tout le fichier comme chaîne de caractères

```
chaine = fd.read()
```

Lire n caractères du fichier comme chaîne de caractères

```
chaine = fd.read(n)
```

Lire une ligne du fichier comme chaîne de caractères

```
ligne = fd.readline()
```

Lire toutes les lignes du fichier dans une liste

```
lignes = fd.readlines()
```

## **Lecture** du fichier (fichier ouvert en mode `"r"`)

Lire le fichier **ligne par ligne** avec une **boucle**

```
for ligne in fd:
    print(ligne)
```

(Équivalent d'une boucle utilisant `fd.readline()`)

La boucle utilise le curseur du fichier de la même manière que les autres fonctions !

**Attention :** Les lignes lues par `readline()`, `readlines()` et la boucle `for` contiennent les sauts de ligne (`"\n"`). Pour les retirer :

```
ligne = fd.readline().rstrip()
```

(`rstrip` retire tous les sauts de lignes, espaces et tabulations à droite de la chaîne de caractères)

## Écriture dans un fichier (fichier ouvert en mode `"w"`, `"a"` ou `"x"`)

Écrire une chaîne de caractères

```
fd.write(chaine)
```

Attention : n'ajoute pas automatiquement le saut de ligne (`"\n"`)

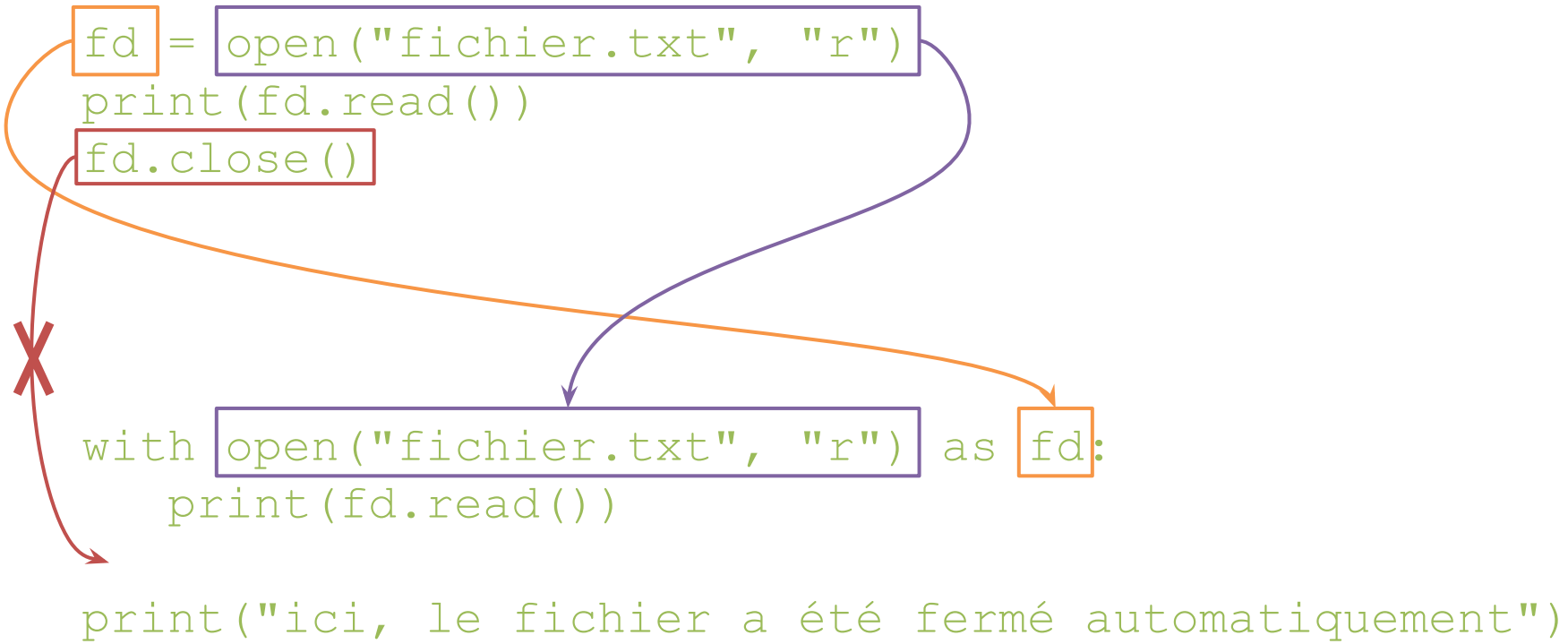
Écrire plusieurs lignes (liste de chaînes de caractère)

```
fd.writelines(liste)
```

Attention : n'ajoute pas non plus les sauts de ligne (`"\n"`)

# Fichiers

Ouverture du fichier avec `with` : **fermeture automatique**




Fichier fermé automatiquement :

- A la fin du bloc
- En cas d'erreur dans le bloc

# Manipulation des chaînes de caractères

**Découper** une chaîne de caractères en **liste d'éléments**, en utilisant un **séparateur** (par défaut, espaces **et** tabulations)

```
liste = chaine.split(separateur)
```

```
animaux = "girafe tigre singe"
ma_liste = animaux.split()
print(ma_liste)
```

=> `["girafe", "tigre", "singe"]`

# Manipulation des chaînes de caractères

**Découper** une chaîne de caractères en **liste d'éléments**, en utilisant un **séparateur** (par défaut, espaces **et** tabulations)

```
liste = chaine.split(separateur)
```

```
animaux = "girafe;tigre;singe"
ma_liste = animaux.split(";")
print(ma_liste)
```

=> `["girafe", "tigre", "singe"]`

# Manipulation des chaînes de caractères

Mettre en **MAJUSCULE/minuscule**

```
chaine_majuscule = chaine.upper()
chaine_mijuscule = chaine.lower()
chaine_capitalisee = chaine.capitalize()
```

**Convertir** une **liste de chaînes de caractères** en **une seule chaîne de caractères**

```
chaine = "séparateur".join(liste)
```

```
liste = ["A", "T", "G", "CAAA"]
seq1 = "-".join(liste)   => "A-T-G-CAA"
seq2 = "".join(liste)    => "ATGCAA"
seq3 = " ".join(liste)   => "A T G CAA"
```

# Manipulation des chaînes de caractères

**Supprimer certains caractères** (donnés dans une chaine de caractères) **des extrémités d'une chaîne de caractères** (par défaut : espaces, tabulations et sauts de ligne)

- des deux côtés de la chaîne :

```
new_chaine = chaine.strip()
```

(paramètres par défaut)

- a l'extrémité droite :

```
new_chaine = chaine.rstrip(".;")
```

(retirera les . et les ;)

- a l'extrémité gauche

```
new_chaine = chaine.lstrip(" \t\n")
```

(retirera les espaces, les tabulations et les sauts de ligne)

# Pratique 6: Les fichiers

## Exercices 9 & 10

Voila une séquence au format fasta (sequence1.fasta)

```
>gi|374429558|ref|NR_046237.1| Rattus norvegicus 18S ribosomal RNA (Rn18s)
TACCTGGTTGATCCTGCCAGTAGCATATGCTTGTCTCAAAGATTAAGCCATGCATGTCTAAGTACGCACG
GCCGGTACAGTGAAACTGCGAATGGCTCATTAAATCAGTTATGGTTCCTTTGGTCGCTCGCTCCTCTCCT
```

- Créer un programme python *exo9.py* qui permet de lire le fichier *sequence1.fasta* (dont le nom est passé en argument) contenant la séquence au format fasta et qui affiche cette séquence

- Créer un programme python *exo10.py* qui permet de lire un fichier (dont le nom est passé en argument) contenant la séquence au format fasta et qui affiche les résidus de la séquence écrits en minuscule (pas l'en-tête fasta ">....").

  **Exemple d'affichage :**

```
tacctggttgatcctgccagtagcatatgcttgtctcaaagattaagccatgcatgtctaagtacgcacg
gccggtacagtgaaactgcgaatggctcattaaatcagttatggttcctttggtcgctcgctcctctcct
acttggataactgtggtaattctagagctaatacatgccgacgggcgctgacccccttcccgtgggggga
...
```

# Pratique 6: Les fichiers

## Exercice 11

Créer un programme python *exo11.py* qui permet de lire un fichier (dont le nom est passé en argument) contenant la séquence au format fasta et qui affiche une ligne sur 2 de la séquence fasta : ligne 1 puis 3 puis 5 …

Exemple d'affichage :

```
>gi|374429558|ref|NR_046237.1| Rattus norvegicus 18S ribosomal RNA (Rn18s)
GCCGGTACAGTGAAACTGCGAATGGCTCATTAAATCAGTTATGGTTCCTTTGGTCGCTCGCTCCTCTCCT
AACGCGTGCATTTATCAGATCAAAACCAACCCGGTCAGCCCCCTCCCGGCTCCGGCCGGGGGTCGGGCGC
```

# Pratique 6: Les fichiers

## Exercice 12

Details de l'en-tête d'une séquence au format fasta (sequence2.fasta)

```
>gi|374429558|ref|NR_046237.1| Rattus norvegicus 18S ribosomal RNA (Rn18s)
```

- Créer un programme python *exo12.py* qui permet de lire un fichier (passé en argument) contenant plusieurs séquences au format fasta et qui affiche les en-têtes de ces séquences.
- Modifier le programme précédent *exo12.py* pour qu'il crée un fichier résultat ne contenant que les en-têtes des séquences fasta (en plus de l'affichage). Le nom du fichier résultat sera passé en argument.

**Exemple de fichier résultat :**

```
>gi|374429558|ref|NR_046237.1| Rattus norvegicus 18S ribosomal RNA (Rn18s)
>gi|374429558|ref|NR_046237.1| Rattus norvegicus 18S ribosomal RNA (Rn18s)
>gi|374429558|ref|NR_046237.1| Rattus norvegicus 18S ribosomal RNA (Rn18s)
>gi|374429558|ref|NR_046237.1| Rattus norvegicus 18S ribosomal RNA (Rn18s)
```

# Pratique 6: Les fichiers

## Exercice 13

Créer un programme python *exo13.py* qui permet de lire un fichier (passé en argument) contenant plusieurs séquences au format fasta et qui crée un fichier résultat (passé en argument) qui devra contenir uniquement le numéro gi et le numéro d'accession de chaque séquence du fichier d'entrée.

**Exemple de fichier résultat :**

```
seq 1 => num gi : 374429558, num accession : NR_046237.1
seq 2 => num gi : 374429558, num accession : NR_046237.1
seq 3 => num gi : 374429558, num accession : NR_046237.1
```

# Pratique 6 : Les fichiers

## Exercice 14

Créer un programme *lecture_tabule.py* prend en argument un fichier tabulé contenant 2 colonnes numériques et écrit un nouveau fichier tabulé en sortie contenant les 2 colonnes originales et une 3ème colonne contenant leur somme

Les noms des fichiers d'entrée et de sortie doivent être pris en argument.

La première ligne du fichier tabulé est un en-tête, il faut juste ajouter le nom de la colonne supplémentaire "Somme".

Un fichier d'entrée d'exemple "`fichier_tabule.tsv`" est disponible dans les données du cours.

**Exemple d'entrée :**

```
Valeur 1    Valeur 2
5           10
3           22
```

**Exemple de sortie :**

```
Valeur 1    Valeur 2 Somme
5           10       15
3           22       25
```

# Dictionnaires

Un **dictionnaire** est une structure de données qui contient une **collection non ordonnée** (pas d'indice) de **couples clé/valeur** (pouvant être de types différents)

**Déclaration**

```
dico = {cle1: valeur1, cle2: valeur2, …}
dico_vide = {}
```

On **accède aux valeurs** d'un dictionnaire par ses clés

```
dico[cle] = valeur
```

# Dictionnaires

Liste* des **clés** d'un dictionnaire : `dico.keys()`

Liste* des **valeurs** d'un dictionnaire : `dico.values()`

Liste* des **couples clé/valeur** d'un dictionnaire : `dico.items()`

```
for cle, valeur in dico.items():
    print(f"La clé {cle} contient {valeur}")
```

* se comportent presque comme des listes mais n'en sont pas tout à fait. On peut obtenir une vraie liste en faisant :

```
list(dico.keys())
```

# Dictionnaires

**Longueur** (nb. de couples) d'un dictionnaire

```
len(dico)
```

note : `len` fonctionne sur beaucoup de types qui ont une longueur : `str`, `list`, `tuple`, `dict`, `set`, `range`...

**Tester l'existence** d'une clé

```
cle in dico
```

**Modifier une valeur** ou **Ajouter un couple**

(selon si la clé existe ou pas)

```
dico[cle] = valeur
```

**Supprimer** un couple

```
del dico[cle]
```

**Vider** un dictionnaire

```
dico.clear()
```

# Pratique 7: Les dictionnaires

## Exercices 15 & 16

- Créer un programme python *exo15.py* qui créer un dictionnaire associant à chaque base de l'ADN la chaine "purinique" (C et G) ou "pyrimidinique" (A et T) selon la base :

```
{'G': 'purinique', 'A': 'pyrimidinique', 'C': 'purinique', 'T': 'pyrimidinique'}
```

  puis qui demande à l'utilisateur (fonction input()) d'entrer une base, et enfin qui affiche si la base entrée par l'utilisateur est purinique ou pyrimidique

- En utilisant un dictionnaire et la fonction in (clef in dictionnaire), créer un programme python exo16.py qui stocke dans un dictionnaire le nombre d'occurrences de chaque acide nucléique de la séquence `"AGWPSGGASAGLAILWGASAIMPGALW"`, puis afficher ce dictionnaire.

  Votre programme doit afficher :

```
{'P': 2, 'I': 2, 'W': 3, 'G': 6, 'L': 3, 'A': 7, 'M': 1, 'S': 3}
```

# Pratique 7: Les dictionnaires

## Exercice 17

- Voila un dictionnaire faisant correspondre à chaque espèce la longueur de son génome :

```
dico_espece = {'Escherichia coli':3.6,'Homo sapiens':3200,'Saccharomyces cerevisae':12,'Arabidopsis thaliana':125}
```

Créer un programme python *exo17.py* dans lequel vous créez ce dictionnaire et qui permet d'afficher le nom de l'organisme possédant le plus grand génome.

# Pratique 7: Les dictionnaires

## Exercice 18

Le fichier *sequences_especes.tsv* est un fichier tabulé (2 colonnes séparées par des tabulations) contenant un identifiant de séquence dans la première colonne, et l'espèce associée dans la secondes colonne.

```
MK032998.1          Homo sapiens
MH180353.1          Homo sapiens
NM_001129758.2   Homo sapiens
SUMJ01000221.1      Escherichia coli
```

Créez un script *exo18.py* lit ce fichier et, à l'aide d'un dictionnaire, écrit un fichier de sortie qui contient le nom d'une espèce précédé d'un dièze sur une ligne puis tous les identifiants de séquence associés à cette espèce sur les lignes suivantes.

```
# Homo sapiens
MK032998.1
MH180353.1
NM_001129758.2
....
# Escherichia coli
SUMJ01000221.1
....
```

# Biopython

Librairie contenant un ensemble d'objets et de fonctions dédiés à la manipulation des données de biologie moléculaire

- **Lecture et écriture** de fichiers (fasta/q, genbank…)
- **Interrogation** directe des **bases de données** (NCBI…)
- **Lancement** simplifié **d'outils** usuels (Blast, Clustal…)

Importer les objets Biopython :

```
from Bio.Seq import Seq, SeqRecord
```
Objets Séquence

```
from Bio import SeqIO
```
Entrée Sortie Séquence

```
from Bio import AlignIO
```
Entrée/Sortie Alignements

# Biopython > Séquences

Objet `Seq` = **séquence** (suite de bases) + **alphabet**

L'alphabet définit le **type** de la séquence : ADN, Protéine, ARN...

```
from Bio import Seq
ma_sequence = Seq("ATTGCGCGAG")
```

Construit un objet Seq

Séquence de l'objet Seq

alphabet non précisé = alphabet générique

Objet `Seq("ATTGCGCGAG")`

# Biopython > Séquences

Objet `Seq` = **séquence** (suite de bases) + **alphabet**

L'alphabet définit le **type** de la séquence : ADN, Protéine, ARN...

```
from Bio import Seq
ma_sequence = Seq("ATTGCGCGAG")
```

Parfois utile de spécifier l'alphabet :

`DNAAlphabet()`, `ProteinAlphabet()`, `RNAAlphabet()`

```
from Bio.Seq import Seq
from Bio.Alphabet import ProteinAlphabet
ma_sequence = Seq("ACGHA", ProteinAlphabet())
```

# Biopython > Séquences

L'object `Seq` se comporte comme une **chaîne de caractères**
(= liste non modifiable)

- Ajout avec `+`

```
ma_sequence = ma_sequence + "TATATA"
```

- Test d'appartenance avec in

```
if "ATG" in ma_sequence
```

Au début et à la fin

```
if ma_sequence.startswith("ATG")
if ma_sequence.endswith("TAA")
```

- Accès aux bases avec [] (index à partir de 0 !)

```
print(ma_sequence[0])
```

# Biopython > Séquences

L'object `Seq` se comporte comme une **chaîne de caractères**
(= liste non modifiable)

- Longueur
  ```
  len(ma_sequence)
  ```

- Position d'une sous-chaîne
  ```
  pos = ma_sequence.find("ATG")
  ```

- Nombre d'occurrences d'un élément (sans superpositions)
  ```
  nombre_a = ma_sequence.count("A")
  ```

# Biopython > Séquences

`Seq` fournit aussi des fonctions spécifiques :

- Complément inverse

```
rev_compl = ma_sequence.reverse_complement()
```

- Traduction

```
ma_proteine = ma_seq.translate()
```

# Biopython > SeqRecord

Objet `SeqRecord` : `Seq` + identifiants + description + informations optionnelles (annotations et features)

```
from Bio.SeqRecord import SeqRecord
```

Attributs :

- `.seq` : objet `Seq`
- `.id` : identifiant
- `.name` : nom commun, par ex. accession
- `.description` : la description
- `.letter_annotations` : dictionnaire des annotations par lettre (qualités…)
- `.annotations` : dictionnaire des annotations (keywords, comments…)
- `.features` : liste de SeqFeature
- `.dbxrefs` : cross-références aux bases de données

# Biopython > SeqRecord

Objet `SeqRecord` : `Seq` + identifiants + description + informations optionnelles (annotations et features)

```
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord
mon_record = SeqRecord(Seq("ATAGC"), id="mySeq",
description="test")
mon_record.name = "maSeq"
```

# Biopython > SeqIO > Lecture

`SeqIO` : Ensemble de fonctions pour la **lecture et écriture de séquences** sous différents formats (fasta/q, GenBank, EMBL…)

```
from Bio import SeqIO
```

Les fonctions de `SeqIO` retournent des objets `SeqRecord`

Lire un fichier avec **UNE** séquence :

```
SeqIO.read(chemin/objet fichier, format)
```

```
from Bio import SeqIO
mon_record = SeqIO.read("seq.fasta", "fasta")
```

Renvoie un `SeqRecord`

# Biopython > SeqIO > Lecture

Lire un fichier avec **PLUSIEURS** séquences :

```
SeqIO.parse(chemin/objet fichier, format)
```

Renvoie un itérateur de `SeqRecord`

```
from Bio import SeqIO
for mon_record in SeqIO
   print(mon_record.id)
```

**attention** : on ne peut parcourir l'itérateur qu'une seule fois

Récupérer toutes les séquences dans une **liste** :

```
liste_records = list(SeqIO.parse(chemin, format)
```

Récupérer toutes les séquences dans un **dictionnaire** dont les clés sont les id des `SeqRecord` :

```
dico_records = SeqIO.to_dict(SeqIO.parse(chemin, format))
```

# Biopython > SeqIO > Écriture

Obtenir une séquence dans un format choisi à partir d'un `SeqRecord` :

```
record.format(format)
```

**Écrire un `SeqRecord` dans un fichier :**

```
fd.write(record.format("fasta"))
```

**Écrire un ensemble de `SeqRecord` dans un fichier :**

```
SeqIO.write(liste_seqrecords, chemin, format)
```

# Biopython > SeqIO > Conversion

**Convertir des séquences d'un format à un autre** directement

```
SeqIO.convert(chemin_in, format_in, chemin_out, format_out)
```

(Renvoie le nombre de séquences converties)

# Pratique 8: bioPython Seq, SeqRecord et SeqIO

## Exercices 19, 20 & 21

On va travailler sur le fichier GenBank *rattus.gb* (dossier de données)

- Créer un programme python *exo19.py* qui prend en argument (paramètre) le fichier de séquences au format Genbank et qui crée un nouveau fichier avec les séquences converties au format fasta.

- Créer un programme python *exo20.py* qui prend en argument (paramètre) le fichier de séquences au format Genbank et qui donne la longueur de chaque séquence en écrivant à l'écran pour chaque séquence : "La séquence ... a pour longueur ... "

- Créer un programme python *exo21.py* qui prend en argument (paramètre) le fichier de séquences au format Genbank et qui crée deux nouveaux fichiers, un avec le complément inverse des séquences au format fasta et un avec les séquences traduites en séquences protéiques au format fasta.

# Biopython > Récupérer des Séquences

Interroger les bases de données et télécharger des séquences

Ici on s'intéresse au BDD du NCBI (ex portail Entrez)

```
from Bio import Entrez
```

**Important :** préciser un email (conditions d'utilisation du NCBI)

```
Entrez.email = "votre@email.fr"
```

# Biopython > Récupérer des Séquences

**1. Effectuer une requête** sur une base de données

```
fic_XML = Entrez.esearch(db=nom_db,
term=requete, …)
```

Renvoie le résultat sous la forme d'un fichier XML ouvert

**2. Parser le résultat** dans un dictionnaire

```
dic_resultat = Entrez.read(fic_XML)
```

Identifiants des séquences : `dico_resultat["IdList"]`

# Biopython > Récupérer des Séquences

**3. Télécharger les séquences** à partir des identifiants

```
fic_sequences = Entrez.efetch(
    db=nom_db,
    id=dic_resultat["IdList"],
    rettype=format
)
```

Renvoie un objet fichier ouvert, à lire avec `SeqIO.read()` ou `SeqIO.parse()`.

4. Penser à **fermer les fichiers** une fois qu'on a fini

```
fic_XML.close()
fic_sequences.close()
```

# Biopython > Documentation

Quelques liens utiles pour utiliser Biopython

Cookbook : tutoriaux sur les différents composants
https://biopython.org/DIST/docs/tutorial/Tutorial.html
https://biopython.org/wiki/Category%3ACookbook

API documentation : documentation complète de tous les objets et fonctions
http://biopython.org/DIST/docs/api/

Un article/tutoriel en français
https://connect.ed-diamond.com/GNU-Linux-Magazine/GLMFHS-073/La-bioinformatique-avec-Biopython

# Pratique 8bis: bioPython, récupération de séquences

## Exercices 22 & 23

- Créer un programme python *exo22.py* qui permet de récupérer dans la banque de données Protein du NCBI les séquences du gène SRY (`SRY [gene]`) de l'homme (`Homo sapiens[Orgn]`) et qui crée un fichier résultat avec les séquences au format Genbank (option `rettype="gb"` de la fonction `efetch()`). Vous limiterez votre recherche aux 10 premières entrées de la banque (option `retmax` de la fonction `esearch()`).

- Créer un programme python *exo23.py* qui permet de faire la même chose que le programme précédent mais qui fait en sorte que le nom de la banque de données du NCBI, le nom du gène et de l'organisme, ainsi que le nom de fichier résultat soient donnés en argument à votre programme.

# Pratique 8bis: bioPython, récupération de séquences

## Exercice 24

- Créer un programme python *exo24.py* qui à partir de 15 séquences d'un gène (donné par l'utilisateur en argument) récupérées sur une banque du NCBI (donnée par l'utilisateur en argument) affiche à l'écran pour chaque séquence : « séquence *nom_sequence* de longueur *lg_sequence* »

- Par exemple voilà l'affichage pour les 3 premières séquences récupérées pour le gène SRY dans la banque Protein du NCBI :

```
sequence  EDW99052  de longeur  530
sequence  EDL09053  de longeur  395
sequence  EAX02769  de longeur  204
```

# Exécuter d'autres programmes : subprocess

On peut vouloir **appeler un programme extérieur** depuis notre script et **récupérer sa sortie**

**! Python >= 3.5**

Module subprocess (`import subprocess`)

Commande à donner sous forme de **liste de chaînes de caractères**, chaque argument étant une nouvelle chaîne.

Ajouter l'option `stdout` pour capturer la sortie et `text` pour automatiquement la décoder (comme à l'ouverture d'un fichier).

Renvoie un objet `CompletedProcess`, la sortie est accessible sur l'argument `.stdout`

# Exécuter d'autres programmes : subprocess

On peut vouloir **appeler un programme extérieur** depuis notre script et **récupérer sa sortie**

**! Python >= 3.5**

Module subprocess (`import subprocess`)

```
import subprocess
commande = ["programme", "argument1", "argument2"...]
process = subprocess.run(
    commande,
    stdout=subprocess.PIPE,
    text=True
)
print(process.stdout)
```

# Exécuter d'autres programmes : subprocess

Arguments de subprocess utiles :

- `stdout=subprocess.PIPE` : récupérer la **sortie standard** du programme

- `check=True` : génère une exception si le programme rencontre une **erreur** (code de sortie ≠ 0)

- `shell=True` : exécuter la commande dans un **shell** (pour utiliser les pipes `|` , les redirections >, …).
  Dans ce cas, la commande doit être une chaîne de caractères.

- `timeout=10` : définit un **délai d'exécution**, en secondes.
  Une erreur se produira si le programme dépasse ce délai.

# Exécuter d'autres programmes : subprocess

Consultez la documentation !

https://docs.python.org/3/library/subprocess.html

# Exercice subprocess

# Bonus : Vos propres fonctions

Écrire vos propres fonctions vous permet de r**endre un bloc de code réutilisable** depuis n'importe où dans votre script, **autant de fois que nécessaire**.

Avantages :

- au lieu de copier coller : on modifie la fonction, cela change le comportement partout où elle est appelée
- lisibilité

# Bonus : Vos propres fonctions

## Syntaxe

```
def nom_de_la_fonction(argument_1, argument2):
    # instructions…
    return valeur_de_retour
```

**Note :** les variables déclarées dans la fonction ne sont pas accessibles depuis l'extérieur (sauf la valeur de retour)

```
def fonction():
    x = 3

print(x)    # Erreur !!
```

# Exemple/Exercice fonction

# Quelques librairies tierces utiles

Interpréteur amélioré

**IPython** https://www.numpy.org/

Notebooks interactifs à la RMarkdown

**jupyter** https://pandas.pydata.org/

# Quelques librairies tierces utiles

Calcul scientifique optimisé (matrices, arrays, …)
**NumPy** https://www.numpy.org/

Manipuler les données sous forme de DataFrames (tableaux)
**pandas** https://pandas.pydata.org/

Génération de graphiques :

- **MatPlotLib** (pyplot) https://matplotlib.org/
- **Plotly** https://plot.ly/python/

Requêtes HTTP (web)
**requests** http://docs.python-requests.org/

# Quelques librairies tierces utiles

Parsing de formats de bioinformatique

FASTA, FASTQ, GB (BLAST, formats d'alignement ésotériques)
**Biopython** https://biopython.org/

SAM, BAM, CRAM (VCF)
**pysam** https://pysam.readthedocs.io/en/latest/

# Formateurs

- Valentin Klein
- Étienne Loire
- Sebastien Ravel



- Valérie Noël
- Julie Orjuela


D’après un support de cours créé par

**Anne-Muriel Chiffoleau** (UM)

# Merci pour votre attention !

# SUIVEZ NOUS SUR TWITTER !

South Green : @green_bioinfo

I-Trop : @ItropBioinfo

# TOGGLe : Bonnes pratiques

## Comment citer les clusters?

"The authors acknowledge the IRD i-Trop HPC at IRD Montpellier for providing HPC resources that have contributed to the research results reported within this paper. URL: http://bioinfo.ird.fr/ "

“The authors acknowledge the CIRAD UMR-AGAP HPC (South Green Platform) at CIRAD montpellier for providing HPC resources that have contributed to the research results reported within this paper. URL: http://www.southgreen.fr”