Introduction à Python

Sandra Dérozier & Thomas Duigou

Déroulé de la formation

- 2 journées : 9h30 - 17h
- 4 pauses de 15 minutes
- 2 pauses déjeuner d'1 heure
    - départ groupé pour la cantine sur le centre
- 20% théorie / 80% pratique
- N'hésitez pas à nous interrompre et à nous poser des questions, nous sommes là pour ça !

Objectifs pédagogiques

- A l'issue de la formation, vous serez capables de :
  - connaître les éléments de base du langage de programmation Python,
  - les appliquer sur des cas concrets en bioinformatique,
  - être autonome dans la mise en place de tâches simples d'extraction d'informations, dans le cadre de traitement de données via le langage de programmation Python.

Programme de la formation

1. Introduction à la programmation
2. Introduction à Python
3. Présentation de Jupyter Notebooks et JupyterLab
4. Variables
5. Affichage
6. Listes, Tuples & Sets
7. Dictionnaires
8. Structures de contrôle
9. Boucles
10. Gestion de fichiers
11. Utilisation de modules

Tour de table

- Comment vous appelez-vous ?
- Quelles sont vos thématiques de recherche ? Sur quoi travaillez-vous ?
- Pourquoi effectuer cette formation ?

Introduction à la programmation

Définition de la programmation informatique

Dans le domaine de l'informatique, la **programmation**, appelée aussi codage, est l'**ensemble des activités** qui permettent l'**écriture des programmes informatiques**. C'est une étape importante du **développement** de logiciels (voire de matériel). Pour **écrire un programme**, on utilise un **langage de programmation**.

Source : wikipédia

Combien y a-t-il de "A" dans cette séquence ?

ATGCTAGCTAGCTAG

Combien y a-t-il de "A" dans cette séquence ?

ATGCTAGCTAGCTAG

ATGCTAGCTAGCTAG

Résultat : il y a 4 adénines dans la séquence.

Combien y a-t-il de "A" dans cette séquence ?

ATGCAGTCGATCGATCGTACGTACGTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGC
TAGCTAGCTAGCTAGCTAGCTAGTCGATCGTAGCTAGCTACGTAGCTAGCTAGCTAG
CTAGCTAGCTAGCTGATCGATCGTAGCTAGCTAGCTAGCTAGCTAGCTACGTAGCTA
GCTAGCTAGCTAGCTAGCTGACTGATCGTAGCTAGCTAGCTAGCTAGCTAGCTAGCT
AGCTAGCTAGCTAGCTAGCTAGCTAGCTGATCGTAGCTAGCTAGCTAGCTAGCTAGC
TGACTAGCTACGTACGTAGCTACGTAGCTAGCTACGTACGTACGTAGCTACGTAGCT
AGCTAGCTGATCGATCGTACGTCGATGCTCGTAGCTAGCTAGCTGATGCTGATGCTA
GCAGCTGTAGCTAGCTAGCTAGCTAGCTAGCTGATCGTAGCTAGCTAGCTAGCTAGC
CTAGCTAGCTAGCTGATCGATCGTAGCTAGCTAGCTAGCTAGCTAGCTACGTAGCTA
AGCTAGCTAGCTAGCTAGCTAGCTAGCTGATCGTAGCTAGCTAGCTAGCTAGCTTAA

Définition d'un algorithme

Un **algorithme** est une **suite finie et non ambiguë d’opérations ou d'instructions** permettant de **résoudre une classe de problèmes**.

Source : wikipédia

Algorithmie

Algorithme humain

ATGCTAGCTAGCTAG

Pour chaque nucléotide de la séquence,

    si le nucléotide est une adénine

        alors on compte une adénine de plus.

Algorithmie

Algorithme humain

ATGCTAGCTAGCTAG

Pour chaque nucléotide de la séquence,

    si le nucléotide est une adénine

        alors on compte une adénine de plus.

Algorithme Python

```
sequence = 'ATGCTAGCTAGCTAG'
nb_A = 0

for nucleotide in sequence:
        if nucleotide == 'A':
                nb_A = nb_A + 1

print(nb_A)
```

Rappel de quelques commandes Unix

Afficher le chemin du répertoire de travail actuel

`pwd` : « print working directory »

```
[sderozier@migale ~]$ pwd
/projet/maiage/save/sderozier
```

Lister le contenu du répertoire de travail

`ls` : « list »

```
[sderozier@migale ~]$ ls
file1.txt   file2.txt   file3.py   file4.sh
```

Création d'un nouveau répertoire

`mkdir` : « make directory »

```
[sderozier@migale ~]$ mkdir NewDirectory
```

Se déplacer vers un autre répertoire

`cd` : « change directory »

```
[sderozier@migale ~]$ cd NewDirectory/
[sderozier@migale NewDirectory]$ pwd
/projet/maiage/save/sderozier/NewDirectory
```

Raccourcis utiles

- ~ représente le répertoire racine (*home directory*) de l'utilsateur
- . représente le répertoire de travail actuel
- .. représente le répertoire *parent* de la localisation actuelle

Déplacer un fichier

`mv` : « move »

```
mv file.txt Directory/
```

Attention :

- si le fichier `file.txt` existe déjà dans `Directory/`, il sera écrasé
- si le caractère `/` est omis, `file.txt` sera renommé en un fichier `Directory` (voir ci-dessous)

Renommer un fichier

```
mv oldname.txt newname.txt
```

Copier un fichier

```
# copier le fichier file.txt dans le répertoire Directory/
cp file.txt Directory/

# copier le fichier name.txt et nommer la copie othername.txt
cp name.txt othername.txt
```

Supprimer un fichier

```
rm file.txt
```

Attention : la suppression est irréversible.

Suppression de répertoire

```
rmdir Directory
```

Pour que la commande s'exécute sans erreur, le répertoire doit être vide.

Éditeurs de texte

Dans le terminal avec Nano

Nano est un éditeur de texte en ligne de commande pour Unix et Linux.

Ouvrir un fichier

```
nano nom_du_fichier.ext
```

Naviguer dans le fichier

Utiliser les touches directionnelles pour se déplacer dans le fichier.

Éditer du texte

- Pour insérer du texte à l'endroit du curseur, il suffit de taper.
- Pour supprimer du texte, utilisez la touche `Suppr` ou `Retour arrière`.

Sauvegarder et quitter

- Pour sauvegarder le fichier : le raccourci clavier est `Ctrl + O`.
- Pour quitter Nano : le raccourci clavier est `Ctrl + X`.

Éditeurs de texte graphiques

Des éditeurs de texte plus puissants et plus conviviaux existent :

- Visual Studio Code (multi-plateforme)
- Atom (multi-plateforme)
- PyCharm (multi-plateforme)
- Gedit (Linux)

Introduction à Python

Qu'est-ce que Python ?

- Création par Guido van Rossum en **1989**
- Langage de programmation **interprété**
    - similaire à Perl, Ruby…
- Adapté au **traitement** et à la **manipulation de fichiers texte**
- Dernière version : **3.12**
- Site officiel : **https://python.org**

Votre environnement de travail

- Activation de l'environnement Conda pour la formation

```
$ conda activate formation-python
```

! Cette instruction est à exécuter à chaque fois que vous ouvrirez une nouvelle session de terminal.

- La documentation de conda est disponible en ligne : https://conda.io/projects/conda/en/latest/index.html.

Exécution de Python

- Interpréteur Python :

```
$ python
Python 3.10.4 (main, Mar 31 2022, 08:41:55) [GCC 7.5.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Remarque : pratique pour tester une fonctionnalité ou exécuter une instruction.

- Exécution d'un script Python :

```
$ python mon_script.py
```

Remarques :

- *l'extension d'un script Python est **.py**.*
- *l'exécution d'un script sera privilégié lors de l'exécution de plusieurs instructions.*

Commentaires & Indentation

- Commentaire :

```
>>> # Ceci est un commentaire car précédé par un '#'
```

- Indentation :

```
>>> instruction1:
...     instruction1_1
...     instruction1_2
...     instruction1_3
>>> instruction2
```

Remarques :

- *l'instruction **instruction1** contient un **bloc d'instructions** constitué de trois instructions identifiables grâce à l'**indentation** du code (soit **4 espaces**, soit une **tabulation**).*
- *l'**instruction 2** est en dehors du bloc d'instructions précédent (absence d'indentation).*

Utilisation de l'interpréteur Python

- Lancement de l'interpréteur Python :

```
$ python
Python 3.10.4 (main, Mar 31 2022, 08:41:55) [GCC 7.5.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- Affichage d'une chaîne de caractères dans l'interpréteur :

```
>>> print("Hello World")
Hello World
```

- Utilisation de l'interpréteur comme une calculatrice :

```
>>> 5+3
8
```

- Quitter l'interpréteur Python :

```
>>> exit()
```

Mon premier script Python

- **Énoncé** : créer un programme Python nommé « `hello.py` » permettant d'afficher les informations demandées en 5 minutes d'autonomie :

Je m'appelle Prénom Nom.

Mon premier script Python

- **Énoncé** : créer un programme Python nommé « `hello.py` » permettant d'afficher les informations demandées en 5 minutes d'autonomie :

Je m'appelle Prénom Nom.

- 1ère solution :

```
print("Je m'appelle Sandra Dérozier.")
```

- 2ème solution :

```
print('Je m\'appelle Sandra Dérozier.')
```

- exécution du script :

```
$ python hello.py
Je m'appelle Sandra Dérozier.
```

Introduction à Jupyter

Qu'est-ce qu'un Jupyter *notebook* ?

- un **cahier électronique** pouvant rassembler du texte, des images, des formules mathématiques, du code informatique, ...
- **manipulable** dans un **navigateur web**
- initialement développé pour Python, R et Julia (supporte actuellement ~40 langages)
- l'élément de base d'un *notebook* Jupyter est la **cellule** qui peut contenir :
  - du **texte** formatté en **Markdown**
  - du **code informatique exécutable**
- site officiel : **https://jupyter.org/**

Votre environnement de travail

- activation de l'environnement Conda pour la formation

```
$ conda activate formation-python
```

- exécution de JupyterLab

```
$ jupyter lab
```

JupyterLab est un environnement complet d'analyse.

Démonstration et premiers pas tous ensemble

Mon premier Jupyter *notebook*

- **Énoncé** : créer un *notebook* Jupyter nommé « `hello.ipynb` » permettant d'afficher les informations demandées en 5 minutes d'autonomie :

```
Je m'appelle Prénom Nom.
```

Variables

Déclaration d'une variable

- opérateur d'affectation "=" :

```
>>> i = 10
>>> i
10
```

- différents types de variable :

```
>>> entier = 5
>>> entier
5
```

```
>>> chaine = 'bonjour'
>>> chaine
'bonjour'
```

```
>>> melange = 'numéro 5'
>>> melange
'numéro 5'
```

Nommage des variables

- un nom de variable **peut contenir** :
    - des lettres minuscules (**a** à **z**)
    - des lettres majuscules (**A** à **Z**)
    - des chiffres (**0** à **9**)
    - le caractère « _ » (*underscore*)
- un nom de variable **ne peut pas** :
    - commencer par un chiffre (**0** à **9**)
    - contenir de caractère accentué
    - contenir de tiret (**-**) ou d'apostrophe (**'**)
    - contenir d'espace
- un nom de variable **ne doit pas être** un mot réservé (exemple : print, else, for, in, ...)
- Python est sensible à la casse : les variables **Titi**, **titi** et **TITI** sont différentes.

Opérations - Types numériques

Les opérateurs sont +, -, *, /, ** et %.

```
>>> i = 5
>>> i + 5
10
>>> i - 2
3
```

```
>>> j = 10
>>> j * 5
50
>>> j / 2
5.0
```

Remarque : la division en Python 3 renvoie un float.

```
>>> k = 3
>>> k ** 2
9
>>> k % 2
1
```

Opérations - Chaines de caractères

Les opérateurs sont + et *.

```
>>> chaine = 'Bonjour'
>>> chaine
'Bonjour'
>>> chaine + ' Madame'
'Bonjour Madame'
```

```
>>> chaine = 'ATGC'
>>> chaine
'ATGC'
>>> chaine * 3
'ATGCATGCATGC'
```

```
>>> 'valeur : ' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not int
```

Remarque : le message d'erreur vous indique qu'il n'est pas possible de concaténer une chaîne de caractères avec un entier.

Booléens

- deux valeurs possibles : **True** et **False**
- trois opérateurs logiques : **and**, **or** et **not**

Expression	Résultat
`True and True`	True
`True and False`	False
`False and True`	False
`False and False`	False

Expression	Résultat
`True or True`	True
`True or False`	True
`False or True`	True
`False or False`	False

Expression	Résultat
`not True`	False
`not False`	True

Opérateurs de comparaison

Les opérateurs de comparaison sont ==, !=, <, <=, > et >=.

```
>>> a = 10
>>> b = 5
>>> a > 5
True
>>> a == b
False
```

```
>>> i = 'bonjour'
>>> j = 'salut'
>>> i == j
False
>>> i < j
True
```

Attention à bien utiliser "==" (tester une égalité) et non "=" (assigner une valeur).

Aller un peu plus loin - Chaines de caractères

- nombre de caractères :

```
>>> sequence = 'ATGGCATGG'
>>> len(sequence)
9
```

- caractère à une position donnée :

```
>>> sequence = 'ATGGCATGG'
>>> sequence[0]
'A'
>>> sequence[3]
'G'
```

- minuscules/majuscules :

```
>>> sequence = 'aTgGcAtgG'
>>> sequence.upper()
'ATGGCATGG'
>>> sequence.lower()
'atggcatgg'
```

Aller un peu plus loin - Chaines de caractères

- récupération d'une sous-chaine (ou tranche) :

```
>>> sequence = 'ATGGCATGG'
>>> sequence[4:7]
'CAT'
```

Remarque : le dernier indice est exclu !

- récupération du début de la chaine :

```
>>> sequence = 'ATGGCATGG'
>>> sequence[:3]
'ATG'
```

- récupération de la fin de la chaine :

```
>>> sequence = 'ATGGCATGG'
>>> sequence[-3:]
'TGG'
```

Aller un peu plus loin - Chaines de caractères

- remplacement de caractères :

```
>>> sequence = 'ATGGCATGG'
>>> sequence.replace('A','T')
'TTGGCTTGG'
```

- vérification du début ou de la fin de la chaine :

```
>>> sequence = 'ATGGCATGG'
>>> sequence.startswith('A')
True
>>> sequence.startswith('T')
False
>>> sequence.endswith('G')
True
```

Fonction Type()

La fonction **Type()** permet de connaître le type d'une variable.

```
>>> i = 5
>>> type(i)
<class 'int'>
```

*Remarque : la variable **i** est ici un **nombre entier**.*

```
>>> j = 0.5
>>> type(j)
<class 'float'>
```

*Remarque : la variable **j** est ici un **nombre réel**.*

```
>>> k = '5'
>>> type(k)
<class 'str'>
```

*Remarque : la variable **k** est ici une **chaine de caractères**.*

Conversion de types

Les fonctions **int()**, **float()** et **str()** permettent de convertir les types.

```
>>> i = 5
>>> type(i)
<class 'int'>
```

*Remarque : la variable **i** est un **nombre entier**.*

```
>>> str(i)
'5'
```

*Remarque : grâce à la fonction **str()**, la variable **i** est convertie en **chaine de caractères**.*

```
>>> float(i)
5.0
```

*Remarque : grâce à la fonction **float()**, la variable **i** est convertie en **nombre réel**.*

Travaux Pratiques

Créer un programme Python nommé **variables.py** ou un *notebook* Jupyter nommé **variables.ipynb** permettant de manipuler cette séquence d'ADN **atGCCcAAaaACcTgGAtAA** et d'afficher les informations suivantes en *25 minutes d'autonomie* :

- la **séquence** en **lettres majuscules uniquement,**
- **la longueur de la séquence,**
- **le nombre de codon,**
- **le premier codon,**
- **le dernier codon.**

Remarque : un codon est une séquence de 3 nucléotides.

Affichage

Affichage d'une ou plusieurs variables

La fonction print() permet l'affichage de variables.

```
>>> nom = 'Sandra'
>>> lieu = 'Jouy-en-Josas'
>>> print(nom, 'travaille à', lieu)
Sandra travaille à Jouy-en-Josas
```

```
>>> prenom = 'Sandra'
>>> nom = 'Dérozier'
>>> print(prenom, nom)
Sandra Dérozier
```

```
>>> prenom = 'Sandra'
>>> nom = 'Dérozier'
>>> print(prenom+nom)
SandraDérozier
```

Écriture formatée

Depuis la version 3.6, Python a introduit les **f-strings**.

f-string est le diminutif de *formatted string literals*.

Exemple

```
f"Ceci est une formatted string literals"
```

Les **f-string** permettent un affichage mieux organisé.

```
>>> nom = 'Sandra'
>>> lieu = 'Jouy-en-Josas'
>>> print(f"{nom} travaille à {lieu}.")
Sandra travaille à Jouy-en-Josas.
```

Remarque : il est possible de mettre également des valeurs numériques entre les {}.

```
>>> num_bat = 233
>>> print(f"{nom} travaille à {lieu}, au bâtiment {num_bat}.")
Sandra travaille à Jouy-en-Josas, au bâtiment 233..
```

Aller plus loin - Écriture formatée

```
>>> i = 34.7879783705980890
>>> print(f"{i}")
34.7879783705980890
>>>
>>> print(f"{i:.2f}")
34.79
```

En détails :

- les **:** indiquent que l'on souhaite préciser un format.
- le **f** précise que l'on souhaite afficher la variable sous la forme d'un réel (*float*).
- le **.2** indique la précision souhaitée, soit deux chiffres après la virgule.

Que modifier si on souhaite plutôt trois chiffres après la virgule ?

Aller plus loin - Écriture formatée

```
>>> i = 34.7879783705980890
>>> print(f"{i}")
34.7879783705980890
>>>
>>> print(f"{i:.2f}")
34.79
```

En détails :

- les **:** indiquent que l'on souhaite préciser un format.
- le **f** précise que l'on souhaite afficher la variable sous la forme d'un réel (*float*).
- le **.2** indique la précision souhaitée, soit deux chiffres après la virgule.

Que modifier si on souhaite plutôt trois chiffres après la virgule ?

```
>>> i = 34.7879783705980890
>>>
>>> print(f"{i:.3f}")
34.788
```

Remarque : le formatage avec `.xf` (x étant un entier positif) renvoie un résultat arrondi.

Aller plus loin - Écriture formatée

Il est également possible de formatter des entiers avec d.

```
>>> annee = 2022
>>> print(f"Nous sommes en {annee:d} !")
Nous sommes en 2022 !
```

Il est également possible de combiner plusieurs nombres dans une même chaîne de caractères.

```
>>> nbG = 3700
>>> nbC = 2450
>>> total = 13800
>>> GCp = (nbG + nbC) / total
>>> print(f"Nombre de G : {nbG:d} / nombre de C : {nbC:d}. \
... Pourcentage en GC de : {(GCp * 100):.2f}%.")
Nombre de G : 3700 / nombre de C : 2450. Pourcentage en GC de : 44.57%.
```

Remarques :

- le symbol \ permet d'écrire une commande sur plusieurs lignes.
- il est possible de mettre des expressions Python dans les {}.

Aller plus loin - Écriture formatée

```
>>> print(11) ; print(1111)
11
1111
```

Remarque : le ; permet de séparer les instructions d'une même ligne.

```
>>> print(f"{11:>10d}") ; print(f"{1111:>10d}")
        11
      1111
```

Les : définissent qu'il va y avoir un formatage, avec un alignement à droite via > (< pour à gauche) sur 10 caractères.

```
>>> print(f"{11:0^10d}") ; print(f"{1111:0^10d}")
0000110000
0001111000
```

Les : définisent qu'il va y avoir un formatage, avec un alignement centré via ^ sur 10 caractères avec un remplissage de 0 (par défaut : espace).

Aller plus loin - Écriture formatée

Ce type de formatage est également possible sur les chaînes de caractères (s).

```
>>> print("gene dnaA") ; print("gene S2")
gene dnaA
gene S2
>>> print(f"gene {dnaA:>4s}") ; print(f"gene {S2:>4s}")
gene dnaA
gene   S2
```

Pour les réels (*float*), on peut utiliser la notation "10.2f" pour 2 chiffres après la virgule formaté sur 10 caractères.

```
>>> propGC = 49.7870787070707
>>> print(f"{propGC:>10.2f}%")
     49.79%
```

Remarque : pour écrire des accolades litérales et utiliser les **f-strings**, il faut doubler les accolades pour échapper au formatage.

Travaux Pratiques

Créer un programme Python nommé **affichage.py** ou un *notebook* Jupyter nommé **affichage.ipynb** permettant d'afficher un triangle composé d'étoiles ayant une base de 9 étoiles via l'écriture formatée en *15 minutes d'autonomie*.

Résultat attendu :

```
    *
   ***
  *****
 *******
*********
```

Remarque : comme nous n'avons pas encore vu les boucles, vous devez écrire toutes les lignes. Vous devez donc avoir 5 lignes de ***print(f"")*** *dans votre code.*

Listes, Tuples & Sets

Listes

Une liste est une structure pour stocker des chiffres, des nombres, des caractères, des chaînes de caractères, ... sous la forme de tableaux.

```
>>> alphabet=['A','B','C','D','E','F','G','H','I','J']
>>> alphabet
['A','B','C','D','E','F','G','H','I','J']
```

Les éléments de la liste sont accessibles par des indices. 'A' est à l'indice 0, 'B' est à l'indice 1, ..., 'J' est à l'indice 9.

```
>>> alphabet[5]
'F' # et non pas 'E'
>>> alphabet[-1]
'J' # dernier élément
```

- Récupération de l'indice

```
>>> lettres = ['a','b','c']
>>> lettres.index('b')
1
```

Listes

- Accès à plusieurs éléments de la liste

```
>>> alphabet[0:3]
['A', 'B', 'C'] # on obtient une sous liste, attention au dernier indice
```

- Nombre d'éléments dans une liste

```
>>> sequence=['a', 'b', 'c', 'd', 'e', 'f']
>>> len(sequence)
6
```

- Nombre d'occurences

```
>>> lettres=['a','b','c','b','b','a']
>>> lettres.count('b')
3
>>> lettres.count('a')
2
```

Listes

- Insérer un élément à une position donnée

```
>>> seq = ['seq2','seq3']
>>> seq.insert(0,'seq1')
>>> seq
['seq1', 'seq2', 'seq3']
```

- Ajouter un élément en fin de liste

```
>>> seq.append('seq4')
>>> seq
['seq1', 'seq2', 'seq3', 'seq4']
```

Listes

- Retrait d'un élément à une position donnée

```
>>> seq = ['seq1','seq2','seq3']
>>> seq.pop(0) # retrait à la position 0
'seq1'
>>> seq # pop modifie la liste
['seq2', 'seq3']
```

- Suppression d'un élément donné

```
>>> seq.remove('seq3')
>>> seq
['seq1', 'seq2']
```

Listes

- Concaténer deux listes

```
>>> seq = ['seq1'] + ['seq2']
>>> seq
['seq1', 'seq2']
```

Remarque : l'opérateur += est un raccourci pour concaténer deux listes.

```
>>> seq += ['seq3']
>>> seq
['seq1', 'seq2', 'seq3']
```

- Transformer une chaîne de caractère en liste grâce à un séparateur

```
>>> sequence = 'a*b*c*d*e*f'
>>> sequence.split('*')
['a', 'b', 'c', 'd', 'e', 'f']
```

Listes

- Passer d'une liste à une chaîne de caractères

```
>>> list_seq = ['A', 'T', 'G', 'C']
>>> list-seq
['A', 'T', 'G', 'C']
>>> seq = "".join(list_seq)
>>> seq
'ATGC'
>>> seq = "-".join(list_seq)
>>> seq
'A-T-G-C'
```

- Duplication d'une liste

```
>>> seq = ['seq1','seq2']
>>> seq*2
>>> seq
['seq1','seq2','seq1','seq2']
```

Listes

- Inverser une liste

```
>>> seq = ['a','b','c']
>>> seq.reverse()
['c','b','a']
```

- Trier dans l'ordre ASCII

```
>>> seq = ['c','b','a']
>>> seq.sort() # sort() ne renvoie rien, trie la liste "sur place"
>>> seq
['a','b','c']
>>> seq = ['c','b','a']
>>> seq2 = sorted(seq) # pour ne pas modifier la liste il faut utiliser sorted()
>>> seq2
['a','b','c']
>>> seq
['c','b','a']
```

Tuples

- Un `tuple` est une liste **non modifiable**.
- Il n'existe donc pas de méthodes pour ajouter/enlever des éléments dans un tuple.
- L'accès aux valeurs par l'indice ainsi que l'interrogation du tuple est possible.
- Exemple

```
>>> t = (1,2,3)
>>> t
(1, 2, 3)
>>> t[1] # accès aux valeurs
2
>>> 2 in t # interroger le tuple
True
```

Sets

Un `set` est **modifiable**, **non ordonné**, **non indexable** et ne contient qu'**une seule copie maximum d'un élément**.

```
>>> s = {1, 2, 3, 4, 5, 6, 1, 2}
>>> s
{1, 2, 3, 4, 5, 6}

>>> type(s)
<class 'set'>
```

```
>>> s[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object is not subscriptable
```

```
>>> s[0] = 9
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support item assignment
```

Remarque : impossible car les sets sont non ordonnés et non indexables.

Sets : opérations

Les sets supportent les opérations d'ensembles. En voici quelques exemples :

- Union

```
>>> s1 = {1, 2, 3}
>>> s2 = {3, 4, 5}
>>> s1 | s2
{1, 2, 3, 4, 5}
```

- Intersection

```
>>> s1 & s2
{3}
```

- Différence

```
>>> s1 - s2
{1, 2}
```

- Différence symétrique

```
>>> s1 ^ s2
{1, 2, 4, 5}
```

Sets : itération

Les sets sont itérables : on peut les parcourir avec une boucle `for`.

```
>>> s = {1, 2, 3, 4, 5, 6}
>>> for elt in s:
...     print(elt)
...
1
2
3
4
5
6
```

Sets : modifications

Les méthodes `add` et `discard` permettent d'ajouter ou supprimer des valeurs d'un `set`.

```
>>> s
{1, 2, 3, 4, 5, 6}
>>> s.add(9)
>>> s
{1, 2, 3, 4, 5, 6, 9}
>>> s.discard(1)
>>> s
{2, 3, 4, 5, 6, 9}
```

L'opérateur `|=` permet d'ajouter des éléments d'un autre `set` à un `set`.

```
>>> s1 = {1, 2, 3}
>>> s2 = {3, 4, 5}
>>> s1 |= s2
>>> s1
{1, 2, 3, 4, 5}
```

Tuples, Listes ou Sets ?

- Les `tuples` sont plus rapides que les `listes`.
- Si vous souhaitez parcourir un ensemble fixe de données, utilisez un `tuple`.
- Utile aussi pour protéger votre code en écriture.

Source :
https://python.developpez.com/cours/DiveIntoPython/php/frdiveintopython/native_data_types/tuples.php

- Les `sets` sont très utiles pour rechercher des éléments uniques dans une suite d'éléments (élimination des doublons).

Travaux Pratiques

Créer un programme Python nommé « `liste.py` » ou un notebook Jupyter nommé « `liste.ipynb` » permettant de manipuler une liste générée à partir de la chaîne de caractères **atg ccc AAA AAC Cat GGa Taa** et d'afficher les informations suivantes en *25 minutes d'autonomie* :

- la **séquence** en **lettres majuscules** uniquement,
- la **longueur** de la séquence,
- le **premier codon**,
- le **dernier codon**.

Dictionnaires

Dictionnaires

- Un dictionnaire est une structure pour stocker des chiffres, des nombres, des caractères, des chaînes de caractères, ...
- Les éléments stockés seront retrouvés à partir d'une clé.
- La clé doit être unique.
- La recherche d'une valeur se fait à partir d'une clé et non l'inverse.

Dictionnaires

- Opérateur d'affectation

Initialiser un dictionnaire vide

```
>>> sequences = {}
```

Initialiser un dictionnaire avec des valeurs

```
>>> sequences = {'seq1':'ATGC', 'seq2':'TGCC'}
>>> sequences
{'seq1': 'ATGC', 'seq2': 'TGCC'}
```

- Ajouter un couple clé - valeur

```
>>> sequences['seq3'] = 'GCTT'
```

Note : si la clé `seq3` existe déjà, sa valeur est remplacée.

- Accéder à une valeur à partir de sa clé

```
>>> sequences['seq3']
'GCTT'
```

Dictionnaires

- Lister les clés

```
>>> sequences.keys()
dict_keys(['seq1', 'seq2', 'seq3'])
```

Le résultat est un objet de type `dict_keys` qui peut être converti en liste.

```
>>> list(sequences.keys())
['seq1', 'seq2', 'seq3']
```

- Lister les valeurs

```
>>> sequences.values()
dict_values(['ATGC', 'TGCC', 'GCTT'])
```

De même pour les valeurs, le résultat est un objet de type `dict_values` qui peut être converti en liste.

```
>>> list(sequences.values())
['ATGC', 'TGCC', 'GCTT']
```

Dictionnaires

- Lister les couples clés - valeurs

```
>>> sequences.items()
dict_items([('seq1', 'ATGC'), ('seq2', 'TGCC'), ('seq3', 'GCTT')])
```

Cas d'utilisation classique (dans une boucle)

Itérer sur les clés, afficher les valeurs associées :

```
>>> for key in sequences:
>>>     print(key, sequences[key])
seq1 ATGC
seq2 TGCC
seq3 GCTT
```

Note : sans autre précision, la boucle itère sur les clés de `sequences`.

Autre cas (itérer directement sur les valeurs)

```
>>> for seq in sequences.values():
>>>     print(seq)
ATGC
TGCC
GCTT
```

Dictionnaires

- Suppression d'un couple clé valeur

```
>>> sequences = {'seq1':'ATGC', 'seq2':'TGCC', 'seq3':'GCTT'}
>>> del sequences['seq2']
>>> sequences
{'seq1': 'ATGC', 'seq3': 'GCTT'}
>>> sequences.pop('seq1') # pop() renvoie directement la valeur
'ATGC'
>>> sequences
{'seq3': 'GCTT'}
```

Travaux Pratiques

Créer un programme Python nommé « `dictionnaire.py` » ou un notebook Jupyter nommé « `dictionnaire.ipynb` » permettant de manipuler un dictionnaire et d'afficher les informations suivantes en *25 minutes d'autonomie* :

- la **séquence** qui a pour nom **A.SE.SE6594**,
- la **liste de tous les noms des séquences** (sans les recopier),
- la **liste de toutes les séquences** (sans les recopier),
- le **liste de tous les noms de séquences rangés par ordre alphabétique**,
- la **longueur de la séquence "A.SE.SE7253"**,
- le **1er codon et le 2ème codon de cette séquence**.

Structures de contrôle

Structure de contrôle

- Les structures de contrôle permettent de contrôler le flux d'exécution d'un programme.
- Syntaxe

```
if condition_1:    # attention à l'indentation
    action_1
elif condition_2:
    action_2
elif condition_3:
    action_3
else:
    action_4
```

- Exemple

```
>>> length = 10
>>> seuil = 15
>>> if length <= seuil:
...     print ("la valeur est inférieure ou égale au seuil fixé")
... else:
...     print ("la valeur est supérieure au seuil fixé")
...
'la valeur est inférieure ou égale au seuil fixé'
```

Tests multiples

- Test de plusieurs conditions simultanément : les opérateurs `and` et `or`
- Exemple

```
>>> x = 2
>>> y = 2
>>> if x == 2 or y == 2:
...     print ("le test est vrai")
...
'le test est vrai'
```

Test de l'existence d'une valeur dans une liste : l'opérateur `in`

- Exemple

```
>>> liste = [1, 2, 3, 4, 5]
>>> if 3 in liste:
...     print ("la valeur 3 est dans la liste")
...
'la valeur 3 est dans la liste'
```

Travaux Pratiques

Créer un programme Python nommé « `controle.py` » ou un notebook Jupyter « `controle.ipynb` » permettant de manipuler le contenu d'un dictionnaire avec des strcutures de contrôle et d'afficher les informations suivantes en *25 minutes d'autonomie* :

- les **séquences** qui ont pour nom **A.KE.Q23-CXC-CG** et **A.SE.SE7535**,
- la **longueur** des 2 séquences précédentes,
- les **premiers codons** des 2 séquences précédentes,
- les **derniers codons** des 2 séquences précédentes,
- **si** les séquences sont identiques,
- **si** les séquences sont de même longeur,
- **si** les premiers codons des 2 séquences précédentes sont bien une **méthionine (ATG)**,
- **si** les **derniers codons** des 2 séquences précédentes sont bien **un codon stop (TAA,TGA ou TAG)**.

Boucles

Boucle For

- Syntaxe

```
>>> for element in variable:
...     action
...
```

- Exemple

```
>>> sequence = ''
>>> nucleotides = ['A', 'T', 'G', 'A', 'C', 'A', 'T']
>>> for nucleotide in nucleotides:
...     # Concaténation des nucléotides
...     sequence += nucleotide
...
>>> print(sequence)
'ATGACAT'
```

Boucle For

- Parcours d'une chaine de caractères

```
>>> sequence = 'ACTGAT'
>>> for nucleotide in sequence:
...     # Parcours caractère par caractère
...     print(nucleotide)
...
'A'
'C'
'T'
'G'
'A'
'T'
```

Boucle For et fonction range()

```
>>> for i in range(3):
...     print(i)
...
0
1
2
```

```
>>> for i in range(3,6):
...     print(i)
...
3
4
5
```

```
>>> for i in range(0,8,2):
...     print(i)
...
0
2
4
6
```

Travaux Pratiques

Créer un programme Python nommé **for.py** ou un *notebook* Jupyter nommé **for.ipynb** permettant de construire une boucle For et d'afficher les informations suivantes en 25 minutes d'autonomie à partir d'un dictionnaire contenant des séquences :

- le **nom** de chaque séquence
- chaque **séquence**
- la **longueur** de chaque séquence
- le 1er codon ; **est-ce que c'est une méthionine ?**
- le dernier codon ; **est-ce que c'est un codon stop ?**

Remarques :

- *méthionine : ATG*
- *codons stop : TAG, TAA, TGA*

Boucle While

- Syntaxe

```
>>> while condition_arret:
...     action
...
```

- Exemple

On a 15mL d'enzyme, chaque digestion a besoin de 5mL, je veux savoir jusqu'à quand je peux lancer mes digestions.

```
>>> vol_enz_depart = vol_enz_restant = 15
>>> vol_enz_reaction = 5
>>> while vol_enz_restant > vol_enz_reaction:
...     # Deduction des 5mL d'enzyme utilisés
...     vol_enz_restant -= vol_enz_reaction
...     print("Vous pouvez lancer votre digestion")
...
'Vous pouvez lancer votre digestion'
'Vous pouvez lancer votre digestion'
```

Break & Continue

L'instruction **break** stoppe la boucle.

```
>>> for i in range(10):
...     if i > 1:
...         break
...     print(i)
...
0
1
```

L'instruction **continue** passe à l'itération suivante.

```
>>> for i in range(3):
...     if i == 1:
...         continue
...     print(i)
...
0
2
```

Travaux Pratiques

Créer un programme Python nommée **while.py** ou un *notebook* Jupyter nommé **while.ipynb** permettant de construire une boucle While et d'afficher les informations suivantes en 25 minutes d'autonomie à partir de la séquence **ATGACTGTAGCTATCGTACGTATGCGTTAA**.

Exemple d'affichage :

```
$ python exo_while.py
Voici le codon numero 1 : ATG
Voici le codon numero 2 : ACT
Voici le codon numero 3 : GTA
…
```

Gestion de fichiers

Lecture du contenu d'un fichier

- Fonction **open** (argument **r** pour *read*) :

```
>>> inputfile = open('fichier.txt', 'r')
```

- Type :

```
>>> inputfile = open('fichier.txt', 'r')
>>> inputfile
<_io.TextIOWrapper name='fichier.txt' mode='r' encoding='UTF-8'>
```

- Contenu du fichier :

```
>>> lignes = inputfile.readlines()
>>> lignes
['ligne1\n', 'ligne2\n']
>>> for ligne in lignes:
...     print(ligne)
ligne1

ligne2

>>> inputfile.close()
```

Ouverture et fermeture propres

Avec **with**, la fermeture du fichier est gérée automatiquement.

```
>>> with open('fichier.txt', 'r') as inputfile:
...     lignes = inputfile.readlines()
...     for ligne in lignes:
...         print(ligne)
...
ligne1

ligne2

>>>
```

Une fois que l'on sort du bloc d'indentation introduit par **with**, la méthode **.close()** est utilisée.

Readlines mais pas que !

```
>>> with open('fichier.txt', 'r') as inputfile:
...     inputfile.read()
...
'ligne1\nligne2\n'
>>>
```

La méthode **read()** lit la totalité du fichier.

```
>>> with open('fichier.txt', 'r') as inputfile:
...     ligne = inputfile.readline()
...     while ligne != "":
...         print(ligne)
...         ligne = inputfile.readline()
...
ligne1

ligne2

>>>
```

La méthode **readline()** lit une ligne d'un fichier et la renvoie. Il faut rappeler la méthode pour lire la ligne suivante.

Qui dit mieux ?

```
>>> with open('fichier.txt', 'r') as inputfile:
...     for ligne in inputfile:
...         print(ligne)
...
ligne1

ligne2

>>>
```

La boucle **for** permet de lire le fichier ligne par ligne. On ajoute souvent l'utilisation de **strip()** qui permet de supprimer les sauts de ligne.

```
>>> with open('fichier.txt', 'r') as inputfile:
...     for ligne in inputfile:
...         ligneClean = ligne.strip()
...         print(ligneClean)
...
ligne1
ligne2
>>>
```

Ecriture dans un fichier

```
>>> contenus = ['ligne1', 'ligne2', 'ligne3']
>>> with open('sortie.txt', 'w') as outputfile:
...     for contenu in contenus:
...         outputfile.write(contenu)
...
```

De la même manière que pour écrire, on peut utiliser **with**. Par contre, on spécifie que l'on est en mode écriture via le **w** pour *write*.

Remarque : on peut également utiliser le mode **a** pour *append* afin d'ajouter du contenu à un fichier existant.

with permet également d'ouvrir plusieurs fichiers en même temps.

```
>>> with open('fichier.txt', 'r') as inputfile, \
... open('sortie.txt', 'w') as outputfile:
...     for ligne in inputfile:
...          outputfile.write('>>> ' + ligne)
```

Travaux Pratiques

Créer un programme Python nommé **fichier.py** ou un *notebook* Jupyter nommé **fichier.ipynb** permettant de transformer le fichier tabulé fichier.txt en un fichier au format FASTA en 25 minutes :

- ouverture en **lecture** du fichier tabulé
- ouverture en **écriture** du fichier FASTA
- **lecture** du fichier tabulé
- **écriture** dans le fichier FASTA

Fichier tabulé :

```
Nom_Sequence    Sequence    Start    Stop    Brin
```

Fichier FASTA :

```
>Nom_Sequence Start..Stop Brin
Sequence
```

Utilisation de modules

Modules

- Les modules sont des programmes Python qui contiennent des fonctions que l'on est amené à utiliser souvent.
- Vous avez accès à une documentation exhaustive sur le site python: https://docs.python.org/3/py-modindex.html
- L'instruction « `import` » permet d'importer l'ensemble des fonctions (ou sous modules) du module:

```
>>> import os  # toutes les fonctions/sous modules du modules os sont disponibles
>>> os.path  # par exemple le sous module path
<module 'posixpath' from '/usr/lib/python3.10/posixpath.py'>
```

Modules

- Avec le mot clé « `from` » il est possible d'importer directement (et uniquement) la fonction/sous module

```
>>> from os import path
>>> path # ici pas de prefix os
<module 'posixpath' from '/usr/lib/python3.10/posixpath.py'>
```

Modules

- Obtenir de l'aide sur un module

```
>>> help(os)
Help on module os:

NAME
    os - OS routines for NT or Posix depending on what system we're on.

MODULE REFERENCE
    http://docs.python.org/3.10/library/os

    The following documentation is automatically generated from the Python
    source files.  It may be incomplete, incorrect or include features that
    are considered implementation detail and may vary between Python
    implementations.  When in doubt, consult the module reference at the ...
```

```
>>> help(os.path)
...
```

- Pour quitter l'aide : « `q` »

Module « os »

Interaction avec le système d'exploitation

```
>>> fullname='/home/maiage/sderozier/scripts/test.py'
>>> import os
>>> help(os.path.dirname)
Help on function dirname in module posixpath:

dirname(p)
    Returns the directory component of a pathname
(END)
>>> os.path.dirname(fullname)
'/home/maiage/sderozier/scripts'
```

```
>>> help(os.path.basename)
Help on function basename in module posixpath:

basename(p)
    Returns the final component of a pathname
(END)
>>> os.path.basename(fullpath)
'test.py'
```

Module « os »

```
>>> help(os.path.abspath)
Help on function abspath in module posixpath:

abspath(path)
    Return an absolute path.
(END)
>>> currentdir = os.path.abspath(".")
>>> currentdir
'/home/oinizan/FORMATION-PYTHON-2018/CORRIGES'
>>> os.listdir(currentdir) # listdir() du module os
['liste.py', 'traduction.py', 'revcomp.py', 'dictionnaire.py', 'controle.py']
```

Module « sys »

La fonction « `argv` » du module « `sys` » permet de récupérer les arguments passés *en ligne de commande* au sein d'une liste.

```
1 # script test.py
2 import sys
3 print(sys.argv)
```

```
$ python test.py alpha beta gamma
['test.py', 'alpha', 'beta', 'gamma']
```

Travaux Pratiques

Créer un programme Python « `prenom-nom.py` » faisant appel au module « `sys` » permettant d'afficher les informations demandées *20 minutes d'autonomie* :

```
Je m'appelle Prénom Nom
```

Le prénom et le nom doivent être passés en argument du programme et récupérés au sein de celui-ci afin d'effectuer l'affichage.

Module « argparse »

Le module « `argparse` » permet une gestion **avancée** des arguments de la ligne de commande.

```
1  # script test-argparse.py
2  import argparse
3
4  parser = argparse.ArgumentParser(description="Description libre.")
5  parser.add_argument("alpha", help="Valeur de alpha")
6  parser.add_argument("beta", help="Valeur de beta")
7  parser.add_argument("--gamma", help="Valeur de gamma")  # Argument optionel
8  args = parser.parse_args()
9
10 print("Valeur de alpha :", args.alpha)
11 print("Valeur de beta :", args.beta)
12 if args.gamma:  # gamma est optionel
13     print("Valeur de gamma ", args.gamma)
```

```
$ python argparse-test.py 1 2 --gamma 3
Valeur de alpha : 1
Valeur de beta : 2
Valeur de gamma  3
```

Module « argparse »

L'argument « `--help` » génère automatiquement un description des arguments du programme :

```
$ python argparse-test.py --help

usage: argparse-test.py [-h] [--gamma GAMMA] alpha beta

Description libre.

positional arguments:
  alpha          Valeur de alpha
  beta           Valeur de beta

options:
  -h, --help     show this help message and exit
  --gamma GAMMA  Valeur de gamma
```

Travaux Pratiques

Créer un nouveau programme Python « `prenom-nom-argparse.py` » qui cette fois utilise le module « `argparse` » pour récupérer les informations `prenom` et `nom` (*20 minutes d'autonomie*).

Exemple d'affichage attendu :

```
Bonjour Prénom Nom, comment allez-vous aujourd'hui ?
```

Exercice complet

Énoncé

Objectif : à partir d'un **répertoire** contenant des fichiers au **format GFF**, **filtrer** les données de ces fichiers en conservant **uniquement les séquences codantes** (CDS) de **taille supérieure ou égale à 600pb mais inférieure ou égale à 1200pb**.

Écrire les fichiers **GFF filtrés** dans un **nouveau répertoire** nommé **GFF_FILTERED**.

Conseils :

- Bien regarder la structure des fichiers
- Procéder étape par étape

Bonus : créer un nouveau fichier tabulé (le séparateur de colonnes est une tabulation) qui résume le contenu des fichiers GFF avec 3 colonnes :

- le nom de chaque souche,
- le nombre de CDS totaux par fichier,
- le nombre de CDS filtrés par fichier.

Données

Extrait du fichier GFF

```
BK006935.2    tpg    CDS    65778    67520    .    -    0
ID=cds28;Parent=rna28;Dbxref=SGD:S000000038,NCBI_GP:DAA06946.1;Name=DAA06946.1;
Note=G1 cyclin involved in cell cycle progression%3B activates Cdc28p kinase to
promote the G1 to S phase transition%3B plays a role in regulating transcription
of the other G1 cyclins%2C CLN1 and CLN2%3B regulated by phosphorylation and
proteolysis%3B acetly-CoA induces CLN3 transcription in response to nutrient
repletion to promote cell-cycle entry.;gbkey=CDS;gene=CLN3;product=cyclin CLN3;
protein_id=DAA06946.1
```

Détail des 9 colonnes :

- Colonne 1 : séquence de référence
- Colonne 2 : source
- Colonne 3 : type d'élément
- Colonne 4 : coordonnée génomique de début de l'élément
- Colonne 5 : coordonnée génomique de fin de l'élément
- Colonne 6 : score
- Colonne 7 : brin
- Colonne 8 : phase
- Colonne 9 : attributs

Données

Extrait du fichier GFF

```
BK006935.2    tpg    CDS    65778    67520    .    -    0
ID=cds28;Parent=rna28;Dbxref=SGD:S000000038,NCBI_GP:DAA06946.1;Name=DAA06946.1;
Note=G1 cyclin involved in cell cycle progression%3B activates Cdc28p kinase to
promote the G1 to S phase transition%3B plays a role in regulating transcription
of the other G1 cyclins%2C CLN1 and CLN2%3B regulated by phosphorylation and
proteolysis%3B acetly-CoA induces CLN3 transcription in response to nutrient
repletion to promote cell-cycle entry.;gbkey=CDS;gene=CLN3;product=cyclin CLN3;
protein_id=DAA06946.1
```

Détail des 9 colonnes :

- Colonne 1 : séquence de référence
- Colonne 2 : source
- **Colonne 3 : type d'élément**
- **Colonne 4 : coordonnée génomique de début de l'élément**
- **Colonne 5 : coordonnée génomique de fin de l'élément**
- Colonne 6 : score
- Colonne 7 : brin
- Colonne 8 : phase
- Colonne 9 : attributs

Quelques pistes

- Comment peut-on récupérer la liste des fichiers d'un répertoire ?
- Quels sont les champs d'intérêt ? Comment les récupérer ?
- De quelles informations a-t-on besoin pour générer la sortie ?

Différentes étapes

- Récupérer la liste des fichiers présents dans le répertoire de travail
- Parcourir l'ensemble de ces fichiers
- Récupérer les informations d'intérêt dans chacun des fichiers
- Filtrer sur le type d'élément (CDS)
- Récupérer les coordonnées génomiques afin de calculer la longueur
- Vérifier la taille de la séquence
- Ecrire la sortie au format attendu