3 Affichage

3.1 La fonction `print()`

Dans le chapitre 1, nous avons rencontré la fonction `print()` qui affiche une chaîne de caractères (le fameux `"Hello world!"`). En fait, la fonction `print()` affiche l'argument qu'on lui passe entre parenthèses **et** un retour à ligne. Ce retour à ligne supplémentaire est ajouté par défaut. Si toutefois, on ne veut pas afficher ce retour à la ligne, on peut utiliser l'argument par « mot-clé » `end` :

```
1  >>> print("Hello world!")
2  Hello world!
3  >>> print("Hello world!", end="")
4  Hello world!>>>
```

Ligne 1. On a utilisé l'instruction `print()` classiquement en passant la chaîne de caractères `"Hello world!"` en argument.

Ligne 3. On a ajouté un second argument `end=""`, en précisant le mot-clé `end`. Nous aborderons les arguments par mot-clé dans le chapitre 9 *Fonctions*. Pour l'instant, dites-vous que cela modifie le comportement par défaut des fonctions.

Ligne 4. L'effet de l'argument `end=""` est que les trois chevrons `>>>` se retrouvent collés après la chaîne de caractères `"Hello world!"`.

Une autre manière de s'en rendre compte est d'utiliser deux fonctions `print()` à la suite. Dans la portion de code suivante, le caractère « `;` » sert à séparer plusieurs instructions Python sur une même ligne :

```
1  >>> print("Hello") ; print("Joe")
2  Hello
3  Joe
4  >>> print("Hello", end="") ; print("Joe")
5  HelloJoe
6  >>> print("Hello", end=" ") ; print("Joe")
7  Hello Joe
```

La fonction `print()` peut également afficher le contenu d'une variable quel que soit son type. Par exemple, pour un entier :

```
1  >>> var = 3
2  >>> print(var)
3  3
```

Il est également possible d'afficher le contenu de plusieurs variables (quel que soit leur type) en les séparant par des virgules :

```
>>> x = 32
>>> nom = "John"
>>> print(nom, "a", x, "ans")
John a 32 ans
```

Python a écrit une phrase complète en remplaçant les variables `x` et `nom` par leur contenu. Vous remarquerez que pour afficher plusieurs éléments de texte sur une seule ligne, nous avons utilisé le séparateur « `,` » entre les différents éléments. Python a également ajouté un espace à chaque fois que l'on utilisait le séparateur « `,` ». On peut modifier ce comportement en passant à la fonction `print()` l'argument par mot-clé `sep` :

```
>>> x = 32
>>> nom = "John"
>>> print(nom, "a", x, "ans", sep="")
Johna32ans
>>> print(nom, "a", x, "ans", sep="-")
John-a-32-ans
```

Pour afficher deux chaînes de caractères l'une à côté de l'autre, sans espace, on peut soit les concaténer, soit utiliser l'argument par mot-clé `sep` avec une chaîne de caractères vide :

```
>>> ani1 = "chat"
>>> ani2 = "souris"
>>> print(ani1, ani2)
chat souris
>>> print(ani1 + ani2)
chatsouris
>>> print(ani1, ani2, sep="")
chatsouris
```

3.2 Écriture formatée

3.2.1 Définitions

Que signifie « écriture formatée » ?

Définition

L'écriture formatée est un mécanisme permettant d'afficher des variables avec un certain format, par exemple justifiées à gauche ou à droite, ou encore avec un certain nombre de décimales pour les *floats*. L'écriture formatée est incontournable lorsqu'on veut créer des fichiers organisés en « belles colonnes » comme par exemple les fichiers PDB (pour en savoir plus sur ce format, reportez-vous à l'annexe A *Quelques formats de données rencontrés en biologie*).

Depuis la version 3.6, Python a introduit les *f-strings* pour mettre en place l'écriture formatée que nous allons décrire en détail dans cette rubrique. Il existe d'autres manières pour formatter des chaînes de caractères qui étaient utilisées avant la version 3.6, nous en avons mis un rappel bref dans la rubrique suivante. Toutefois, nous conseillons vivement l'utilisation des *f-strings* si vous débutez l'apprentissage de Python.

Que signifie *f-string* ?

Définition

f-string est le diminutif de *formatted string literals*. Mais encore ? Dans le chapitre précédent, nous avons vu les chaînes de caractères ou encore *strings* qui étaient représentées par un texte entouré de guillemets simples ou doubles. Par exemple :

```
"Ceci est une chaîne de caractères"
```

L'équivalent en *f-string* est tout simplement la même chaîne de caractères précédée du caractère `f` **sans espace** entre les deux :

```
f"Ceci est une chaîne de caractères"
```

Ce caractère `f` avant les guillemets va indiquer à Python qu'il s'agit d'une *f-string* permettant de mettre en place le mécanisme de l'écriture formatée, contrairement à une *string* normale.

Nous expliquons plus en détail dans le chapitre 10 *Plus sur les chaînes de caractères* pourquoi on doit mettre ce `f` et le mécanisme sous-jacent.

3.2.2 Prise en main des *f-strings*

Les *f-strings* permettent une meilleure organisation de l'affichage des variables. Reprenons l'exemple ci-dessus à propos de notre ami John :

```
>>> x = 32
>>> nom = "John"
>>> print(f"{nom} a {x} ans")
John a 32 ans
```

Il suffit de passer un nom de variable au sein de chaque couple d'accolades et Python les remplace par leur contenu ! Première remarque, la syntaxe apparait plus lisible que l'équivalent vu ci-avant `print(nom, "a", x, "ans")`. Bien sûr, il ne faut pas omettre le `f` avant le premier guillemet, sinon Python prendra cela pour une chaîne de caractères normale et ne mettra pas en place ce mécanisme de remplacement :

```
>>> print("{nom} a {x} ans")
{nom} a {x} ans
```

Remarque

Une variable est utilisable plus d'une fois pour une *f-string* donnée :

```
>>> var = "to"
>>> print(f"{var} et {var} font {var}{var}")
to et to font toto
>>>
```

Enfin, il est possible de mettre entre les accolades des valeurs numériques ou des chaînes de caractères :

```
>>> print(f"J'affiche l'entier {10} et le float {3.14}")
J'affiche l'entier 10 et le float 3.14
>>> print(f"J'affiche la chaine {'Python'}")
J'affiche la chaine Python
```

Même si cela ne présente que peu d'intérêt pour l'instant, il s'agit d'une commande Python parfaitement valide. Nous verrons des exemples plus pertinents par la suite. Cela fonctionne avec n'importe quel type de variable (entiers, chaînes de caractères, *floats*, etc.). Attention toutefois pour les chaînes de caractères, utilisez des guillemets simples au sein des accolades si vous définissez votre *f-string* avec des guillemets doubles.

3.2.3 Spécification de format

Les *f-strings* permettent de remplacer des variables au sein d'une chaîne de caractères. On peut également spécifier le format de leur affichage.

Prenons un exemple. Imaginez maintenant que vous vouliez calculer, puis afficher, la proportion de GC d'un génome. La proportion de GC s'obtient comme la somme des bases Guanine (G) et Cytosine (C) divisée par le nombre total de bases (A, T, C, G) du génome considéré. Si on a, par exemple, 4500 bases G et 2575 bases C, pour un total de 14800 bases, vous pourriez procéder comme suit (notez bien l'utilisation des parenthèses pour gérer les priorités des opérateurs) :

```
>>> prop_GC = (4500 + 2575) / 14800
>>> print("La proportion de GC est", prop_GC)
La proportion de GC est 0.4780405405405405
```

Le résultat obtenu présente trop de décimales (seize dans le cas présent). Pour écrire le résultat plus lisiblement, vous pouvez spécifier dans les accolades `{}` le format qui vous

intéresse. Dans le cas présent, vous voulez formater un *float* pour l'afficher avec deux puis trois décimales :

```
>>> print(f"La proportion de GC est {prop_GC:.2f}")
La proportion de GC est 0.48
>>> print(f"La proportion de GC est {prop_GC:.3f}")
La proportion de GC est 0.478
```

Détaillons le contenu des accolades de la première ligne (`{prop_GC:.2f}`) :

- D'abord on a le nom de la variable à formatter, `prop_GC`, c'est indispensable avec les *f-strings*.
- Ensuite on rencontre les deux-points `:`, ceux-ci indiquent que ce qui suit va spécifier le format dans lequel on veut afficher la variable `prop_GC`.
- À droite des deux-points on trouve `.2f` qui indique ce format : la lettre `f` indique qu'on souhaite afficher la variable sous forme d'un *float*, les caractères `.2` indiquent la précision voulue, soit ici deux chiffres après la virgule.

Notez enfin que le formatage avec `.xf` (`x` étant un entier positif) renvoie un résultat arrondi.

Vous pouvez aussi formater des entiers avec la lettre `d` (ici `d` veut dire *decimal integer*) :

```
>>> nb_G = 4500
>>> print(f"Ce génome contient {nb_G:d} guanines")
Ce génome contient 4500 guanines
```

ou mettre plusieurs nombres dans une même chaîne de caractères.

```
>>> nb_G = 4500
>>> nb_C = 2575
>>> print(f"Ce génome contient {nb_G:d} G et {nb_C:d} C, soit une prop de GC de {prop_GC:.2f}")
Ce génome contient 4500 G et 2575 C, soit une prop de GC de 0.48
>>> perc_GC = prop_GC * 100
>>> print(f"Ce génome contient {nb_G:d} G et {nb_C:d} C, soit un %GC de {perc_GC:.2f} %")
Ce génome contient 4500 G et 2575 C, soit un %GC de 47.80 %
```

Enfin, il est possible de préciser sur combien de caractères vous voulez qu'un résultat soit écrit et comment se fait l'alignement (à gauche, à droite ou centré). Dans la portion de code suivante, le caractère `;` sert de séparateur entre les instructions sur une même ligne :

```
>>> print(10) ; print(1000)
10
1000
>>> print(f"{10:>6d}") ; print(f"{1000:>6d}")
    10
  1000
```

```
>>> print(f"{10:<6d}") ; print(f"{1000:<6d}")
10
1000
>>> print(f"{10:^6d}") ; print(f"{1000:^6d}")
  10
 1000
>>> print(f"{10:*^6d}") ; print(f"{1000:*^6d}")
**10**
*1000*
>>> print(f"{10:0>6d}") ; print(f"{1000:0>6d}")
000010
001000
```

Notez que `>` spécifie un alignement à droite, `<` spécifie un alignement à gauche et `^` spécifie un alignement centré. Il est également possible d'indiquer le caractère qui servira de remplissage lors des alignements (l'espace est le caractère par défaut).

Ce formatage est également possible sur des chaînes de caractères avec la lettre `s` (comme *string*) :

```
>>> print("atom HN") ; print("atom HDE1")
atom HN
atom HDE1
>>> print(f"atom {'HN':>4s}") ; print(f"atom {'HDE1':>4s}")
atom   HN
atom HDE1
```

Vous voyez tout de suite l'énorme avantage de l'écriture formatée. Elle vous permet d'écrire en colonnes parfaitement alignées. Nous verrons que ceci est très pratique si l'on veut écrire les coordonnées des atomes d'une molécule au format PDB (pour en savoir plus sur ce format, reportez-vous à l'annexe A *Quelques formats de données rencontrés en biologie*).

Pour les *floats*, il est possible de combiner le nombre de caractères à afficher avec le nombre de décimales :

```
>>> print(f"{perc_GC:7.3f}")
 47.804
>>> print(f"{perc_GC:10.3f}")
    47.804
```

L'instruction `7.3f` signifie que l'on souhaite écrire un *float* avec 3 décimales et formaté sur 7 caractères (par défaut justifiés à droite). L'instruction `10.3f` fait la même chose sur 10 caractères. Remarquez que le séparateur décimal `.` compte pour un caractère. De même, si on avait un nombre négatif, le signe `-` compterait aussi pour un caractère.

3.2.4 Autres détails sur les *f-strings*

Si on veut afficher des accolades littérales avec les *f-strings*, il faut les doubler pour échapper au formatage :

```
>>> print(f"Accolades littérales {{}} ou {{ ou }} et pour le formatage {10}")
Accolades littérales {} ou { ou } et pour le formatage 10
```

Une remarque importante, si on ne met pas de variable à formater entre les accolades dans une *f-string*, cela conduit à une erreur :

```
>>> print(f"accolades sans variable {}")
  File "<stdin>", line 1
SyntaxError: f-string: empty expression not allowed
```

Enfin, il est important de bien comprendre qu'une *f-string* est indépendante de la fonction `print()`. Si on donne une *f-string* à la fonction `print()`, Python évalue d'abord la *f-string* et c'est la chaîne de caractères qui en résulte qui est affichée à l'écran. Tout comme dans l'instruction `print(5*5)`, c'est d'abord la multiplication (`5*5`) qui est évaluée, puis son résultat qui est affiché à l'écran. On peut s'en rendre compte de la manière suivante dans l'interpréteur :

```
>>> f"{perc_GC:10.3f}"
'    47.804'
>>> type(f"{perc_GC:10.3f}")
<class 'str'>
```

Python considère le résultat de l'instruction `f"{perc_GC:10.3f}"` comme une chaîne de caractères et la fonction `type()` nous le confirme.

3.2.5 Expressions dans les *f-strings*

Une fonctionnalité extrêmement puissante des *f-strings* est de supporter des expressions Python au sein des accolades. Ainsi, il est possible d'y mettre directement une opération ou encore un appel à une fonction :

```
>>> print(f"Le résultat de 5 * 5 vaut {5 * 5}")
Le résultat de 5 * 5 vaut 25
>>> print(f"Résultat d'une opération avec des floats : {(4.1 * 6.7)}")
Résultat d'une opération avec des floats : 27.47
>>> print(f"Le minimum est {min(1, -2, 4)}")
Le minimum est -2
>>> entier = 2
>>> print(f"Le type de {entier} est {type(entier)}")
Le type de 2 est <class 'int'>
```

Nous aurons l'occasion de revenir sur cette fonctionnalité au fur et à mesure de ce cours.

Les possibilités offertes par les *f-strings* sont nombreuses. Pour vous y retrouver dans les différentes options de formatage, nous vous conseillons de consulter ce mémo (en anglais).

3.3 Écriture scientifique

Pour les nombres très grands ou très petits, l'écriture formatée permet d'afficher un nombre en notation scientifique (sous forme de puissance de 10) avec la lettre `e` :

```
>>> print(f"{1_000_000_000:e}")
1.000000e+09
>>> print(f"{0.000_000_001:e}")
1.000000e-09
```

Il est également possible de définir le nombre de chiffres après la virgule. Dans l'exemple ci-dessous, on affiche un nombre avec aucun, 3 et 6 chiffres après la virgule :

```
>>> avogadro_number = 6.022_140_76e23
>>> print(f"{avogadro_number:.0e}")
6e+23
>>> print(f"{avogadro_number:.3e}")
6.022e+23
>>> print(f"{avogadro_number:.6e}")
6.022141e+23
```

3.4 Ancienne méthode de formatage des chaînes de caractères

Conseil : Pour les débutants, tout ce qui est écrit dans cette rubrique n'est pas à retenir.

Dans les premières versions de Python jusqu'à la 2.6, il fallait utiliser l'opérateur `%`, puis de la version 2.7 jusqu'à la 3.5 il était plutôt conseillé d'utiliser la méthode `.format()` (voir la rubrique suivante pour la définition du mot « méthode »). Même si les *f-strings* sont devenues la manière conseillée pour mettre en place l'écriture formatée, ces deux anciennes manières, sont encore pleinement compatibles avec les versions modernes de Python.

Même si elle fonctionne encore, la première manière avec l'opérateur `%` est maintenant clairement déconseillée pour un certain nombre de raisons. Néanmoins, nous rappelons ci-dessous son fonctionnement, car il se peut que vous tombiez dessus dans d'anciens livres ou si vous lisez de vieux programmes Python.

La deuxième manière avec la méthode `.format()` est encore largement utilisée et reste tout à fait valide. Elle est clairement plus puissante et évite un certain nombre de désagréments par rapport à l'opérateur `%`. Vous la croiserez sans doute très fréquemment dans des programmes et ouvrages récents. Heureusement elle a un fonctionnement relativement proche des *f-strings*, donc vous ne serez pas totalement perdus !

Enfin, nous indiquons à la fin de cette rubrique nos conseils sur quelle méthode utiliser.

3.4.1 L'opérateur `%`

On a vu avec les entiers que l'opérateur `%` ou *modulo* renvoyait le reste d'une division entière. Cet opérateur existe aussi pour les chaînes de caractères mais il met en place l'écriture formatée. En voici un exemple :

```
>>> x = 32
>>> nom = "John"
>>> print("%s a %d ans" % (nom, x))
John a 32 ans
>>> nb_G = 4500
>>> nb_C = 2575
>>> prop_GC = (nb_G + nb_C)/14800
>>> print("On a %d G et %d C -> prop GC = %.2f" % (nb_G, nb_C, prop_GC))
On a 4500 G et 2575 C -> prop GC = 0.48
```

La syntaxe est légèrement différente. Le symbole `%` est d'abord appelé dans la chaîne de caractères (dans l'exemple ci-dessus `%d`, `%d` et `%.2f`) pour :

- Désigner l'endroit où sera placée la variable dans la chaîne de caractères.
- Préciser le type de variable à formater, `d` pour un entier (`i` fonctionne également) ou `f` pour un *float*.
- Éventuellement pour indiquer le format voulu. Ici `.2` signifie une précision de deux décimales.

Le signe `%` est rappelé une seconde fois (`% (nb_G, nb_C, prop_GC)`) pour indiquer les variables à formater.

3.4.2 La méthode `.format()`

Depuis la version 2.7 de Python, la méthode `.format()` (voir la rubrique suivante pour la définition d'une méthode) a apporté une nette amélioration pour mettre en place l'écriture formatée. Celle-ci fonctionne de la manière suivante :

```
>>> x = 32
>>> nom = "John"
>>> print("{} a {} ans".format(nom, x))
John a 32 ans
>>> nb_G = 4500
>>> nb_C = 2575
>>> prop_GC = (nb_G + nb_C)/14800
>>> print("On a {} G et {} C -> prop GC = {:.2f}".format(nb_G, nb_C,
prop_GC))
On a 4500 G et 2575 C -> prop GC = 0.48
```

- Dans la chaîne de caractères, les accolades vides `{}` précisent l'endroit où le contenu de la variable doit être inséré.
- Juste après la chaîne de caractères, l'instruction `.format(nom, x)` fournit la liste des variables à insérer, d'abord la variable `nom` puis la variable `x`.

- On peut éventuellement préciser le formatage en mettant un caractère deux-points `:` puis par exemple ici `.2f` qui signifie 2 chiffres après la virgule.
- La méthode `.format()` agit sur la chaîne de caractères à laquelle elle est attachée par le point.

Tout ce que nous avons vu avec les *f-strings* sur la manière de formatter l'affichage d'une variable (après les `:` au sein des accolades) est identique avec la méthode `.format()`. Par exemple `{:.2f}`, `{:0>6d}`, `{:.6e}`, etc., fonctionneront de la même manière. La différence notable est qu'on ne met pas directement le nom de la variable au sein des accolades. Comme pour l'opérateur `%`, c'est l'emplacement dans les arguments passés à la méthode `.format()` qui dicte quelle variable doit être remplacée. Par exemple, dans `"{} {} {}".format(bidule, machin, truc)`, les premières accolades remplaceront la variable `bidule`, les deuxièmes la variable `machin`, les troisièmes la variable `truc`.

Le formattage avec la méthode `.format()` se rapproche de la syntaxe des *f-strings* (accolades, deux-points), mais présente l'inconvénient -- comme avec l'opérateur `%` -- de devoir mettre la liste des variables tout à la fin, alourdissant ainsi la syntaxe. En effet, dans l'exemple avec la proportion de GC, la ligne équivalente avec une *f-string* apparait tout de même plus simple à lire :

```
>>> print(f"On a {nb_G} G et {nb_C} C -> prop GC = {prop_GC:.2f}")
On a 4500 G et 2575 C -> prop GC = 0.48
```

Conseils

Pour conclure, ces deux anciennes façons de formater une chaîne de caractères avec l'opérateur `%` ou la méthode `.format()` vous sont présentées à titre d'information. La première avec l'opérateur `%` est clairement déconseillée. La deuxième avec la méthode `.format()` est encore tout à fait valable. Si vous débutez Python, nous vous conseillons fortement d'apprendre et d'utiliser les *f-strings*. C'est ce que vous rencontrerez dans la suite de ce cours. Si vous connaissez déjà Python et que vous utilisez la méthode `.format()`, nous vous conseillons de passer aux *f-strings*. Depuis que nous les avons découvertes, aucun retour n'est envisageable pour nous tant elles sont puissantes et plus claires à utiliser !

Enfin, si vous souhaitez aller plus loin, voici deux articles en anglais très bien fait sur le site *RealPython*: sur l'écriture formatée et sur les *f-strings*

3.5 Note sur le vocabulaire et la syntaxe

Revenons quelques instants sur la notion de **méthode** abordée dans ce chapitre avec `.format()`. En Python, on peut considérer chaque variable comme un objet sur lequel on peut appliquer des méthodes. Une méthode est simplement une fonction qui utilise et/ou agit sur

l'objet lui-même, les deux étant connectés par un point. La syntaxe générale est de la forme `objet.méthode()`.

Dans l'exemple suivant :

```
>>> "Joe a {} ans".format(20)
'Joe a 20 ans'
```

la méthode `.format()` est liée à `"Joe a {} ans"` qui est un objet de type chaîne de caractères. La méthode renvoie une nouvelle chaîne de caractères avec le bon formatage (ici, `'Joe a 20 ans'`).

Nous aurons de nombreuses occasions de revoir cette notation `objet.méthode()`.

3.6 Exercices

Conseil : utilisez l'interpréteur Python pour les exercices 2 à 5.

3.6.1 Affichage dans l'interpréteur et dans un programme

Ouvrez l'interpréteur Python et tapez l'instruction `1+1`. Que se passe-t-il ?

Écrivez la même chose dans un script `test.py` que vous allez créer avec un éditeur de texte. Exécutez ce script en tapant `python test.py` dans un *shell*. Que se passe-t-il ? Pourquoi ? Faites en sorte d'afficher le résultat de l'addition `1+1` en exécutant le script dans un *shell*.

3.6.2 Poly-A

Générez une chaîne de caractères représentant un brin d'ADN poly-A (c'est-à-dire qui ne contient que des bases A) de 20 bases de longueur, sans taper littéralement toutes les bases.

3.6.3 Poly-A et poly-GC

Sur le modèle de l'exercice précédent, générez en une ligne de code un brin d'ADN poly-A (AAAA...) de 20 bases suivi d'un poly-GC régulier (GCGCGC...) de 40 bases.

3.6.4 Écriture formatée

En utilisant l'écriture formatée, affichez en une seule ligne les variables `a`, `b` et `c` dont les valeurs sont respectivement la chaîne de caractères `"salut"`, le nombre entier `102` et le *float* `10.318`. La variable `c` sera affichée avec 2 décimales.

3.6.5 Écriture formatée 2

Dans un script `percGC.py`, calculez un pourcentage de GC avec l'instruction suivante :

```
perc_GC = ((4500 + 2575)/14800)*100
```

Ensuite, affichez le contenu de la variable `perc_GC` à l'écran avec 0, 1, 2 puis 3 décimales sous forme arrondie en utilisant l'écriture formatée et les *f-strings*. On souhaite que le programme affiche la sortie suivante :

```
Le pourcentage de GC est 48     %
Le pourcentage de GC est 47.8   %
Le pourcentage de GC est 47.80  %
Le pourcentage de GC est 47.804 %
```