Notes de cours
Semaine 3

Cours Turing

1 Les collections en Python

Ces premières semaines de cours, nous avons vu comment concevoir des programmes plus ou moins simples en Python. Les programmes que vous avez conçus jusqu'à présent travaillaient sur des valeurs relativement simples, comme par exemple des nombres entiers ou encore des chaînes de caractères. C'est en manipulant ces différentes valeurs que vos programmes réalisent les différentes tâches qu'on veut leur faire accomplir. Cette semaine, nous allons enrichir nos connaissances en Python en introduisant d'autres types de valeurs très intéressants, les *collections*.

En termes très généraux, une collection est simplement une valeur qui contient d'autres valeurs. Suivant son type, une collection pourra représenter une séquence ordonnée de valeurs ou encore un ensemble sans ordre particulier, pour ne citer que deux exemples.

Prenons un exemple concret avec un type de collection particulièrement utile en Python, les *listes*. Une liste représente une séquence ordonnée de valeurs. En Python, il est possible de créer des listes et de faire des opérations sur ces listes, comme, entre autres, compter son nombre d'éléments ou accéder à un élément à une certaine position. Le programme ci-dessous montre comment il est possible de créer une liste de prénoms et d'effectuer des opérations sur cette liste.

```
noms = ["Elsa", "Anna", "Olaf", "Kristoff", "Sven"]
print(len(noms))  # Affiche le nombre d'éléments de la liste
print(noms[0])  # Affiche le premier nom de la liste
```

Dans l'exemple ci-dessus, on remarque que l'on a pu effectuer des opérations sur la liste des valeurs, comme par exemple la stocker dans une variable, calculer sa taille ou encore accéder à un de ses éléments. Dans le reste de ces notes de cours, nous allons aborder différentes collections en Python et discuter des différentes opérations qui sont proposées sur ces collections.

1.1 Tuple

La première collection que nous allons aborder est celle des *tuples*. En Python, un tuple représente une séquence ordonnée de valeurs, potentiellement de différents types.

Construction Pour former un tuple, on note simplement, entre parenthèses, les différentes expressions séparées par des virgules. Suivant le contexte, les parenthèses sont optionnelles.

```
("Boxe", "Natation", "Tennis")
```

Les tuples peuvent contenir aussi des valeurs de type différents, et même d'autres collections. Notez aussi que l'on peut donner des expressions plus complexes que de simples valeurs littérales.

```
("Jessica", 3 * 7, 170, ("Photographie", "Esca" + "lade"))
```

Déconstruction Il n'est pas rare en Python, étant donné un tuple, de vouloir affecter chaque élément à une variable. Pour ce faire, on peut lister sur la gauche du signe = lors d'une assignation non pas une seule variable, mais une séquence de variables séparées par des virgules.

```
personne = ("Albert", 19, 180, ("Foot", "Échecs"))  # Construction
prenom, age, taille, hobbies = personne  # Déconstruction
print(age)  # Affiche 19
```

Accès aux éléments Pour accéder directement à un élément d'un tuple, on indique la position de l'élément entre deux crochets ([et]).

```
("A", "B", "C")[0]  # A pour valeur "A"
```

La position du premier élément est 0, celle du deuxième élément 1, et ainsi de suite. Comme l'on commence par la position 0, le dernier élément d'un tuple de longueur n a pour position $n - 1$.

```
("A", "B", "C")[2]  # A pour valeur "C"
```

Il est aussi possible d'indiquer une position négative. Dans ce cas, les positions sont comptées depuis la fin, -1 étant le dernier élément, -2 l'avant-dernier, etc.

```
lettres = ("A", "B", "C")
lettres[-1]  # A pour valeur "C"
lettres[-2]  # A pour valeur "B"
```

Notez aussi que la position à accéder peut aussi être spécifiée par une expression plus complexe qu'un nombre entier littéral.

```
k = 2
("A", "B", "C")[1 + k - 3]  # A pour valeur "A"
```

Lorsque l'on tente d'accéder à une position qui n'existe pas dans la liste, Python lance une erreur de type `IndexError`.

Sous-parties La même syntaxe que pour l'accès à un élément permet de référer à une sous-séquence du tuple. Dans ce cas, on utilisera non pas une seule position, mais deux positions séparées par un symbole deux-points (:).

```
("A", "B", "C", "D", "E")[2:4]  # A pour valeur ("C", "D")
```

Le premier nombre indique la position de début de la sous-partie. La valeur à cette position est incluse dans le résultat. Le deuxième nombre indique la position de fin de la sous-partie. Contrairement à la position de début, la valeur à la position de fin n'est pas incluse. Les

positions de début et de fin peuvent être omises. Lorsque la valeur de début est omise, la sous-partie commence au début de la liste. Lorsque la valeur de fin est omise, la sous-partie termine à la fin de la liste.

Longueur On appelle le nombre d'éléments d'un tuple sa longueur. La longueur est accessible via la fonction `len`.

Parcours des éléments Grâce à la fonction `len` et à la possibilité d'accéder à des éléments via leur position, il est possible, à l'aide d'une boucle `while`, de passer en revue tous les éléments d'un tuple et ainsi d'exécuter des instructions pour chaque élément.

```
fruits = ("pommes", "bananes", "poires")
i = 0
while i < len(fruits):
    print("J'aime les", fruits[i])
    i += 1
```

Cependant, cela n'est pas la façon de faire recommandée en Python. En Python, on préfèrera l'utilisation d'une boucle `for`, comme ceci :

```
fruits = ("pommes", "bananes", "poires")
for fruit in fruits:
    print("J'aime les", fruit)
```

Le code est plus concis que la version avec `while`. De plus, il est facile de faire des erreurs avec la version `while`, en oubliant, par exemple, d'incrémenter la variable ou même de la déclarer. Pour cette raison, on préférera utiliser des boucles `for` quand cela sera possible.

Parcours avancés Parfois, on souhaitera tout de même avoir accès à la position de l'élément courant dans une boucle `for`. Pour ce faire, on utilisera la fonction `enumerate`, comme ceci :

```
fruits = ("pommes", "bananes", "poires")
for i, fruit in enumerate(fruits):
    print(i, fruit)  # Affichera 1 pommes, 2 bananes, puis 3 poires
```

Parfois, on voudra parcourir les éléments dans l'ordre inverse. Pour cela, on pourra utiliser la fonction `reversed`.

```
fruits = ("pommes", "bananes", "poires")
for fruit in reversed(fruits):
    print(fruit)  # Affichera poires puis bananes puis pommes
```

D'autres fois, on voudra parcourir les éléments du plus petit au plus grand. Pour cela, on utilisera la fonction `sorted`.

```
fruits = ("pommes", "bananes", "poires")
for fruit in sorted(fruits):
    print(fruit)  # Affichera bananes puis poires puis pommes
```

Parfois, on voudra parcourir plusieurs séquences de valeurs en même temps. Si le but est de passer en revue toutes les combinaisons possibles de valeurs, on utilisera des boucles imbriquées.

```
fruits = ("pommes", "bananes", "poires")
couleurs = ("rouges", "jaunes", "vertes")
for fruit in fruits:
    for couleur in couleurs:
        print(fruit, couleur)  # Affichera toutes les combinaisons
```

Si, au contraire, on cherche à parcourir en parallèle deux ou plusieurs séquences, on utilisera la fonction `zip`, comme dans l'exemple suivant.

```
fruits = ("pommes", "bananes", "poires")
couleurs = ("rouges", "jaunes", "vertes")
for fruit, couleur in zip(fruits, couleurs):
    print(fruit, couleur)  # Affichera pommes rouges
                           # puis bananes jaunes
                           # puis poires vertes
```

Concaténation L'opérateur `+` permet de *concaténer* deux tuples pour former un unique tuple. Le tuple résultant de la concaténation contient les valeurs du premier tuple suivies des valeurs du deuxième tuple.

```
t1 = (1, 2)
t2 = (3, 4)
t1 + t2  # A pour valeur (1, 2, 3, 4)
```

Appartenance L'opérateur `in` permet de tester l'appartenance d'une valeur à un tuple, comme dans l'exemple ci-dessous.

```
ps = (2, 3, 5, 7)
print(3 in ps)  # Affiche True
print(4 in ps)  # Affiche False

print(3 not in ps)  # Affiche False
print(4 not in ps)  # Affiche True
```

Le temps de calcul d'une opération `in` dépend linéairement du nombre d'éléments du tuple.

1.2 Listes

Toutes les opérations que l'on a vues sur les tuples sont applicables de la même manière sur les listes. À priori, la seule différence est qu'à la place d'utiliser les parenthèses pour former une liste, on utilise les crochets.

```
["Elsa", "Anna", "Olaf", "Kristoff", "Sven"]
```

Pourquoi donc faire une distinction entre tuples et listes en Python ? La réponse vient de ce qu'on appelle la *mutabilité*.

Mutabilité À la différence des tuples, les listes sont *modifiables*. On dit qu'elles sont *mutables*. Prenons un exemple.

```
noms = ["Elsa", "Anna", "Olaf", "Kristoff", "Sven"]
noms.pop()  # Supprime le dernier élément
print(len(noms))  # Affiche 4
noms.pop()  # Supprimer le dernier élément
print(len(noms))  # Affiche 3
print(noms)  # Affiche ["Elsa", "Anna", "Olaf"]
```

Dans l'exemple ci-dessus, la liste stockée dans la variable `noms` voit son contenu modifié par l'exécution de la méthode `pop`, et ce même que la variable `noms` n'est pas modifiée. Notez qu'il n'y a aucune réassignation de la variable, et pourtant la liste a bel et bien changé.

Par le passé, lorsque nous voulions modifier une valeur, nous passions par une réaffectation de la variable qui contenait la valeur. La valeur en soi n'était pas modifiée, seulement la variable. La valeur stockée était simplement remplacée par une autre valeur.

```
n = 3
print(n)  # Affiche 3
n = 4
print(n)  # Affiche 4
print(3)  # Affiche 3
```

Dans l'exemple ci-dessus, les valeurs en soi ne changent pas, ce sont les variables qui changent. Au début, la variable `n` prend pour valeur 3, puis ensuite elle prend pour valeur 4. La valeur 3 quant à elle ne change pas. 3 reste 3 et ne devient pas 4.

Dans le cas des listes, et des autres valeurs mutables que l'on pourra aborder, la situation sera différente : la valeur *elle-même* pourra être modifiée.

Aliasing Ce phénomène de modification de valeurs sans modification de variables peut parfois amener à des situations difficiles à comprendre, surtout lorsque la même valeur est accessible sous différents noms (ce que l'on nomme *aliasing*).

```
xs = [1, 2, 3]
ys = xs

print(ys)  # Affiche [1, 2, 3]
xs.pop()  # Enlève le dernier élément de la valeur stockée dans xs
print(ys)  # Affiche [1, 2]
```

Dans l'exemple ci-dessous, la valeur stockée dans la variable `ys` a changé en cours d'exécution alors qu'un élément a été retiré de la valeur stockée dans `xs`. La situation s'explique facilement car il s'agit *de la même valeur* qui est stockée à la fois dans `xs` et dans `ys`. Pour éviter cette situation, il est possible de faire une *copie* de la liste.

Copie d'une liste Parfois, il sera utile, comme dans le cas discuté juste à l'instant, avec la problématique d'aliasing, de faire une copie d'une liste. Une copie est une liste en tout point identique mais qui a sa propre identité. Les opérations faites sur une liste n'affectent pas les éventuelles copies de la liste. Pour copier une liste en Python, on appelle la méthode `copy`.

```
xs = [1, 2, 3]
ys = xs.copy()  # Stocke une copie de xs dans ys

print(ys)  # Affiche [1, 2, 3]
xs.pop()  # Enlève le dernier élément de la valeur stockée dans xs
print(xs)  # Affiche [1, 2]
print(ys)  # Affiche [1, 2, 3]
ys.pop()
print(xs)  # Affiche [1, 2]
```

Dans l'exemple ci-dessus, on observe que la liste originale et sa copie ne sont pas affectées par les modifications effectuées sur l'autre liste.

Opérations sur les listes

Maintenant que nous avons mis en lumière ce phénomène de mutabilité, passons en revue les différentes opérations qui modifient le contenu d'une liste.

Modification des éléments Grâce à une assignation, il est possible de remplacer la valeur stockée à une certaine position dans la liste. Pour cela, sur la partie de gauche du `=`, on utilisera la même syntaxe que pour accéder à l'élément.

```
animaux = ["chats", "chiens", "poissons"]
animaux[2] = "oiseaux"
print(animaux)  # Affiche ["chats", "chiens", "oiseaux"]
```

Ajout d'éléments à la fin La méthode `append` permet d'ajouter un élément à la fin d'une liste.

```
xs = [1, 2, 3]
xs.append(4)
print(xs)  # Affiche [1, 2, 3, 4]
```

La méthode `append` permet d'ajouter un unique élément à la fin. Pour ajouter plus d'un élément à la fois, on utilisera la méthode `extend`. La méthode `extend` permet d'ajouter une séquence d'éléments à la fin d'une liste.

```
xs = [1, 2, 3]
ys = [4, 5, 6]
xs.extend(ys)
print(xs)  # Affiche [1, 2, 3, 4, 5, 6]
print(ys)  # Affiche [4, 5, 6]
```

Notez au passage que la liste de valeurs passée en argument à `extend` n'est elle pas modifiée.

Ajout d'éléments à des positions arbitraires La méthode `insert` permet d'insérer un élément à une position donnée dans la liste.

```
xs = ["A", "B", "D", "E"]
xs.insert(2, "C")
print(xs)  # Affiche ["A", "B", "C", "D", "E"]
```

Il est cependant important de noter que d'insérer un élément au milieu d'une liste est une opération beaucoup plus complexe pour Python que d'ajouter un élément à la fin. Alors que l'ajout d'un élément à la fin d'une liste prend un temps constant [1] (qui ne dépend pas de la taille de la liste), ajouter un élément au milieu de la liste demande à Python de déplacer tous les éléments qui suivent la position d'insertion. Cette opération prend, dans le pire des cas, un temps linéaire dans le nombre d'éléments de la liste.

Suppression d'éléments à la fin La méthode `pop` permet de retirer le dernier élément d'une liste. La méthode retourne l'élément ainsi retiré.

```
xs = [1, 2, 3]
print(xs.pop())  # Affiche 3
print(xs.pop())  # Affiche 2
print(xs.pop())  # Affiche 1
print(xs)  # Affiche []
```

Suppression d'éléments à des positions arbitraires La méthode `pop` peut aussi s'utiliser pour retirer un élément à une position quelconque de la liste. Pour cela, on ajoute comme argument à l'appel à `pop` la position à laquelle retirer une valeur.

```
xs = [1, 2, 3]
print(xs.pop(1))  # Affiche 2
print(xs.pop(0))  # Affiche 1
print(xs.pop(0))  # Affiche 3
print(xs)  # Affiche []
```

Tout comme pour l'ajout, la suppression d'éléments à la fin d'une liste est plus rapide qu'à des positions arbitraires de la liste.

Trier une liste On appelle trier une liste le fait de mettre les éléments de la liste dans l'ordre, généralement du plus petit au plus grand. Pour trier une liste en Python, on utilise la méthode `sort`.

```
xs = [4, 3, 5, 1, 2]
xs.sort()
print(xs)  # Affiche [1, 2, 3, 4, 5]
```

1. En moyenne. Certains ajouts peuvent prendre plus de temps, mais la moyenne est constante. On parle de complexité temporelle amortie.

1.3 Ensembles

En Python, on appelle un `set` (un ensemble) une collection non-ordonnée de valeurs. Les ensembles en Python, contrairement aux tuples et aux listes, n'ont pas de notion d'ordre. Il n'y a pas à proprement parler de premier ou de dernier élément d'un `set`. De plus, les ensembles ne permettent pas de compter combien de fois un élément fait partie de l'ensemble. Soit la valeur fait partie de l'ensemble, soit elle ne fait pas partie de l'ensemble. On ne peut pas compter *combien* de fois elle fait partie de l'ensemble.

Tout comme les listes, les ensembles en Python sont mutables. Il sera possible de modifier les valeurs contenues dans un `set`.

Les ensembles sont souvent utilisés lorsque l'on s'intéresse à l'appartenance de valeurs plutôt qu'à leur ordre ou à leur nombre d'appartenances.

Construction Pour construire un ensemble, on utilise la fonction `set`. La fonction prend en argument optionnel une collection de valeurs, comme une liste ou un tuple, et forme un ensemble de ces éléments. Lorsque l'on ne fournit pas d'argument à `set`, un ensemble vide est retourné.

```
xs = set([1, 2, 3]) # Ensemble des valeurs 1, 2, 3
ys = set()  # Nouvel ensemble vide.
zs = {1, 2, 3}  # Autre syntaxe (quand non-vide)
```

Appartenance Tout comme pour les listes et les tuples, l'opérateur `in` permet de tester l'appartenance dans un ensemble.

```
xs = set([1, 2, 3]) # Ensemble des valeurs 1, 2, 3
print(2 in xs)  # Affiche True
print(5 in xs)  # Affiche False
```

Contrairement aux listes, le temps d'exécution d'une opération `in` sur un ensemble prend un temps en principe constant.

Ajout d'éléments La méthode `add` permet d'ajouter un élément dans un ensemble.

```
xs = set([2, 3])
xs.add(1)
print(xs)  # Affiche {1, 2, 3}
xs.add(2)
print(xs)  # Affiche {1, 2, 3}
```

La méthode `update` permet d'ajouter une collection d'éléments à un ensemble.

```
xs = set([2, 3])
xs.update([1, 3, 4])
print(xs)  # Affiche {1, 2, 3, 4}
```

Suppression d'éléments La méthode `remove` permet de retirer un élément de l'ensemble. Si l'élément n'existe pas, une erreur est levée.

```
xs = set([2, 3])
xs.remove(2)
print(xs)  # Affiche {3}
xs.remove(5)  # Lance une erreur
```

La méthode `discard` permet elle aussi de retirer un élément d'un ensemble, mais ne soulève pas d'erreur lorsque l'élément ne fais pas partie du `set`.

```
xs = set([2, 3])
xs.discard(2)
print(xs)  # Affiche {3}
xs.discard(5)  # Pas de problème
```

Opérations ensemblistes La plupart des opérations ensemblistes dont vous avez peut-être l'habitude (union, intersection, différence) peuvent être retrouvées sous forme de méthodes de `set`. On trouve la méthode `union` pour l'union, `intersection` pour l'intersection et `difference` pour la différence. Ces méthodes existent sous une forme simple qui retourne simplement le résultat de l'opération sans modifier la valeur, mais aussi sous une forme qui modifie la valeur de l'ensemble sur lequel la méthode est appelée. Le nom de ces dernières méthodes consiste simplement en le nom de la méthode simple suivi de `_update` (sauf pour `union`, où la méthode s'appelle simplement `update`). Par exemple, on aura la méthode `intersection` et la méthode `intersection_update`.

```
xs = set([1, 2])
ys = set([2, 3])
print(xs.union(ys))  # Affiche {1, 2, 3}
print(xs)  # Affiche {1, 2}
print(ys)  # Affiche {2, 3}

xs.intersection_update(ys)
print(xs)  # Affiche {2}
print(ys)  # Affiche {2, 3}
```

Sous-ensemble La méthode `issubset` permet de déterminer si un ensemble est un sous-ensemble d'un autre. De façon symétrique, la méthode `issuperset` permet de déterminer si un ensemble est un super-ensemble d'un autre.

```
xs = set([1, 2, 3])
ys = set([1, 2, 3, 4, 5])
print(xs.issubset(ys))  # Affiche True
print(ys.issuperset(xs))  # Affiche True
```

Cardinalité La cardinalité d'un ensemble en Python, c'est-à-dire sa quantité d'éléments, s'obtient grâce à la méthode `len`.

1.4 Dictionnaires

Finalement, la dernière collection que nous allons aborder aujourd'hui est *le dictionnaire*. Un dictionnaire en Python est un moyen d'associer certaines valeurs, appelée des clés, à d'autres valeurs. On parle parfois de *tableau associatif* ou encore de *table d'association*, bien qu'en Python l'on préfère le terme de dictionnaire. Les dictionnaires sont mutables et n'ont pas de notion d'ordre entres les éléments.

Construction Pour construire un dictionnaire en Python, on utilise des accolades ({ et }). Entre ces accolades, on liste les associations clé/valeur séparées par des virgules. Chaque association est notée en écrivant d'abord la clé, puis le symbole deux-points (`:`), puis la valeur associée. Voyez plutôt.

```
{"nom": "Alain", "age": 16, "taille": 165}  # Construction
```

On note le dictionnaire vide simplement `{}`.

Appartenance d'une clé On peut vérifier si une clé appartient au dictionnaire en utilisant l'opérateur `in`.

```
dettes = {"Doris": 50, "Bastien": 12}
print("Doris" in dettes)  # Affiche True
print("Jean" in dettes)  # Affiche False
```

Accéder à une valeur Pour accéder à la valeur associée à une clé, on utilise la même syntaxe que pour l'accès à un élément d'une liste ou d'un tuple. Si la clé n'existe pas, une erreur de type `KeyError` est lancée.

```
dettes = {"Doris": 50, "Bastien": 12}
print(dettes["Doris"])  # Affiche 50
print(dettes["Jean"])  # Donne lieu à une erreur
```

La méthode `get` permet d'accéder à la valeur associée à une clé sans se soucier de savoir si la clé existe. La méthode prend un deuxième argument en plus de la clé qui indique la valeur à retourner en cas de clé absente.

```
dettes = {"Doris": 50, "Bastien": 12}
print(dettes.get("Doris", 0))  # Affiche 50
print(dettes.get("Jean", 0))  # Affiche 0
```

Ajouter ou modifier une valeur Pour ajouter une valeur ou pour modifier une valeur associée à une clé, on utilise la même opération, celle de l'assignation. À gauche du signe `=`, on note simplement l'élément comme si on y accédait.

```
dettes = {"Doris": 50, "Bastien": 12}
dettes["Marc"] = 25  # Ajout d'une entrée
dettes["Doris"] = 40  # Modification d'une entrée
print(dettes)  # Affiche {"Doris": 40, "Bastien": 12, "Marc": 25}
```

Suppression d'élément Pour supprimer une valeur, on utilise la méthode `pop`. La méthode prend en argument la clé à supprimer et retourne la valeur (anciennement) associée.

```
dettes = {"Doris": 50, "Bastien": 12}
print(dettes.pop("Bastien"))  # Affiche 12
print(dettes)  # Affiche {"Doris": 50}
```

Clés et valeurs d'un dictionnaire La méthode `keys` permet d'itérer sur les clés d'un dictionnaire.

```
dettes = {"Doris": 50, "Bastien": 12}
for key in dettes.keys():
    print(key)  # Affichera Doris, puis Bastien
```

De manière similaire, la méthode `values` permet d'itérer sur les valeurs contenues dans le dictionnaire.

```
dettes = {"Doris": 50, "Bastien": 12}
for value in dettes.values():
    print(value)  # Affichera 50, puis 12
```

Finalement, pour itérer sur les associations clé/valeur du dictionnaire, on utilisera la méthode `items`.

```
dettes = {"Doris": 50, "Bastien": 12}
for key, value in dettes.items():
    print(key, value)  # Affichera Doris 50, puis Bastien 12
```