On découvre :

- comment répéter l'exécution de certaines instructions sur une série de valeurs
- la notion d'itérable, structure qu'il est possible de parcourir
- comment parcourir une séquence de valeurs telle une chaîne de caractères

Dans certaines situations, on peut vouloir exécuter les mêmes instructions sur une série de valeurs –tels des caractères ou des nombres–. Il s'agit par exemple d'afficher ces valeurs, ou vérifier que des caractères sont des lettres, ou vérifier que des entiers sont pairs, *etc.*

Il nous faut pouvoir identifier :

- la série de valeurs : ce sera un *itérable*, notion que nous allons découvrir
- les instructions à exécuter pour chaque valeur de la série : nous utiliserons la boucle `for` de Python

1 Syntaxe

La structure de contrôle appelée boucle `for` permet la répétition d'un bloc d'instructions sur chacune des valeurs d'un itérable.

```
for <variable_iteration> in <iterable> :
    <bloc des instructions qui seront répétées>
```

Notons les mots-clés :

- `for` qui initie la boucle, et
- `in` qui introduit l'*itérable*,

ainsi que le caractère `:` suivi de lignes indentées qui délimitent le bloc d'instructions qui est exécuté au sein de la boucle.

2 Exemple

On utilise ci-dessous le prédicat `str.isupper()` qui renvoie `True` ssi la chaîne passée en paramètre contient uniquement des majuscules.

On utilise aussi les fonctions `str.upper()` et `str.lower()` qui se comportent ainsi :

```
>>> 'a'.upper()
'A'
>>> 'A'.lower()
'a'
```

On considère l'exemple suivant, qui code une version restreinte de la fonction `swapcase` de Python.

```
def inverser_casse(chaine:str) -> str:
    """ Renvoie une chaîne qui contient les lettres du paramètre
    dans le même ordre, mais en inversant la casse majuscule/minuscule.

    Précondition :chaine contient uniquement des lettres de a..z ou A..Z
    Exemple(s) :
    $$$ inverser_casse('AeR')
    'aEr'
    $$$ inverser_casse('ADN')
    'adn'
    $$$ inverser_casse('sel')
    'SEL'
    $$$ inverser_casse('')
    ''
    """
    res = ''
    for car in chaine:
        if car.isupper():
            res = res + car.lower()
        else:
            res = res + car.upper()
    return res
```


Le corps de la fonction contient une boucle `for`. On repère :

- la variable d'itération `car`
- l'itérable `chaine`
- le bloc d'instructions à répéter :

```
if car.isupper():
    res = res + car.lower()
else:
    res = res + car.upper()
```

Ce bloc contient une conditionnelle à la semaine 4 et se lit :

- si `car` est une majuscule
- alors concaténer à `res` la minuscule correspondante,
- sinon (c'est-à-dire que `car` est une minuscule) concaténer à `res` la majuscule correspondante.

Le corps de la fonction se lit ainsi :

- initialement la variable `res` qui contiendra le résultat de la fonction est associée à la chaîne vide ;
- pour chaque caractère `car` de `chaine` (`for car in chaine:`), effectuer le traitement suivant :
  - si `car` est une majuscule
  - alors concaténer à `res` la minuscule correspondante,
  - sinon (c'est-à-dire que `car` est une minuscule) concaténer à `res` la majuscule correspondante
- renvoyer la valeur de `res`.

3 Sémantique et vocabulaire

3.1 Initialisation

La ou les instructions qui précèdent la boucle `for` réalisent souvent une phase d'*initialisation*.

L'initialisation a pour but d'associer une première valeur (appelée *valeur initiale*) aux variables qui seront utilisées dans le corps de la boucle.

Dans notre exemple, l'initialisation `res = ''` indique que le futur résultat de la fonction prend initialement la valeur d'une chaîne vide.

La valeur de `res` sera ensuite modifiée dans le corps de la boucle `for` par l'exécution de l'une ou l'autre des affectations à `res`.

3.2 Itérable

Un *itérable* est une *séquence de valeurs* que l'on peut parcourir.

Dans l'exemple, l'itérable est la chaîne de caractères associée à la variable `chaine`. Toutes les chaînes de caractères sont des itérables, car ce sont des séquences de caractères.

On peut aussi écrire :

```
for car in "chat":
   ...
```

Dans ce cas l'itérable est le littéral `"chat"`, composé de 4 caractères.

Nous découvrirons d'autres itérables par la suite.

3.3 Variable d'itération, itération

La *variable d'itération* d'une boucle `for` est toujours définie entre les mots-clés `for` et `in`.

Dans l'exemple, la variable d'itération est `car`.

Lors de l'exécution de la boucle `for`:

- la variable d'itération prend successivement chacune des valeurs présentes dans l'itérable
- le bloc d'instructions dans le corps du `for` est exécuté avec chacune de ces valeurs successives. Chaque exécution de ce bloc est appelée une *itération* ou un *tour de boucle*.

Quand un itérable contient n valeurs, la boucle effectue n itérations (ou répétitions, ou tours de boucle).

3.4 Exemple d'exécution avec 3 itérations

Si on prend l'appel `inverser_casse('AeR')` : la variable `chaine` est associée à la valeur `'AeR'`, contenant 3 caractères. La boucle effectuera donc 3 itérations. Voici comment se déroule l'appel.

Lors de l'appel, la mémoire contient :

`chaine`	`'AeR'`

Après l'initialisation de `res`, la mémoire contient :

`chaine`	`'AeR'`
`res`	`''`

Ensuite la boucle s'exécute et réalise successivement les itérations qu'on peut numéroter par 1, 2 et 3.

Itération 1 : exécution du corps du `for` avec `car` valant `'A'`

- `car` prend la valeur `'A'`, premier caractère de l'itérable.
- `car.isupper()` vaut `True`
- l'affection `res = res + car.lower()` est exécutée, la valeur de `car.lower()` est `'a'`, `res` prend donc la valeur `'a'`.

À la fin de cette itération la mémoire contient donc :

`chaine`	`'AeR'`
`res`	`'a'`
`car`	`'A'`

Itération 2 : exécution du corps du `for` avec `car` valant `'e'`, en se basant sur la mémoire laissée par l'itération précédente :

- `car` prend la valeur `'e'`, 2ème caractère de l'itérable.
- `car.isupper()` vaut `False`
- l'affection `res = res + car.upper()` est exécutée, la valeur de `car.upper()` est `'E'`, `res` prend donc la valeur `'aE'`.

À la fin de cette itération la mémoire contient donc :

`chaine`	`'AeR'`
`res`	`'aE'`
`car`	`'e'`

Itération 3 : exécution du corps du `for` avec `car` valant `'R'`, en se basant sur la mémoire laissée par l'itération précédente :

- `car` prend la valeur `'R'`, 3ème et dernier caractère de l'itérable.
- `car.isupper()` vaut `True`
- l'affection `res = res + car.lower()` est exécutée, la valeur de `car.lower()` est `'r'`, `res` prend donc la valeur `'aEr'`.

À la fin de cette itération la mémoire contient donc :

`chaine`	`'AeR'`
`res`	`'aEr'`
`car`	`'R'`

Les 3 valeurs de l'itérable ont été traitées : l'exécution de la boucle `for` termine.

L'appel termine avec la valeur `'aEr'`.

3.5 Autre exemple d'exécution sans itération

Si on prend l'appel `inverser_casse('')` : la variable `chaine` est associée à la chaîne vide, contenant 0 caractère. On parle d'*itérable vide*.

La boucle effectuera donc 0 itération. Voici comment se déroule l'appel.

Après l'initialisation de `res`, la mémoire contient :

`chaine`	`''`
`res`	`''`

La boucle `for` n'effectue aucune itération, c'est-à-dire que le bloc d'instruction interne à la boucle n'est pas du tout exécuté. La boucle termine de suite.

L'appel termine avec la valeur `''`.

4 À propos de la variable d'itération

4.1 Portée

La portée de la variable d'itération n'est pas limitée à la boucle `for`.

À l'issue de l'exécution d'une boucle, la variable existe toujours et contient la valeur du dernier élément de l'itérable.

```
>>> chaine = "AeR"
>>> res = ''
    for car in chaine:
        if car.isupper():
            res = res + car.lower()
        else:
            res = res + car.upper()
>>> car
'R'
>>> res
'aEr'
```

4.2 Absence d'initialisation

On notera que dans la fonction `inverser_casse` la variable d'itération n'apparaît pas dans la phase d'initialisation. La boucle `for` est chargé d'attribuer à la variable d'itération ses valeurs successives.

Si on écrit une initialisation avec une valeur donnée, la boucle `for` écrasera cette valeur.

Dans cet exemple, on copie dans `res` les caractères d'une chaîne donnée.

```
>>> res = ''
>>> car = 'Z'
>>> for car in "chat":
       res = res + c
>>> res
"chat"
```

La valeur `'Z'` a bien été perdue.

4.3 Absence d'affectation

On notera que dans la fonction `inverser_casse` le corps de la boucle ne contient aucune affectation à la variable d'itération. Elle contient juste des utilisations de sa valeur.

Quand la boucle `for` est exécutée, la valeur de l'itérable est évaluée avant toute chose, ce qui détermine :

- les valeurs successives prises par la variable d'itération
- et de là le nombre d'itérations.

Si on adopte une *mauvaise pratique* et qu'on modifie la valeur de la variable d'itération dans le corps de la boucle, c'est tout de même la valeur calculée avant l'exécution des itérations qui sera utilisée.

Dans cet exemple on voit que la valeur `'Z'` n'apparaît pas du tout dans la valeur finale de `res` :

```
>>> res = ''
>>> for c in "chat":
       res = res + c
       c = 'Z'
>>> res
"chat"
```

De même il ne sert à rien de modifier la valeur de l'itérable en cours d'exécution de la boucle.

```
>>> res = ''
>>> chaine = 'chat'
>>> for c in chaine:
       res = res + c
       chaine = ''
>>> res
'chat'
```

4.4 Bonnes pratiques de nommage

L'identificateur choisi pour la variable d'itération doit être choisi avec soin, en fonction de l'itérable auquel il est associé. Les bonnes pratiques de nommage seront précisées lors de la présentation des différents types d'itérables.

Avec un itérable de type chaîne de caractères, la variable d'itération peut prendre le nom `c` pour signifier qu'elle va contenir un **c**aractère, ou `car`, `char`, `carac`… mais pas `x` ni `i` ni `a`.

5 Memento

- un *itérable* est une séquence de valeurs qui peut être parcourue à l'aide d'une boucle `for`
- les chaînes de caractères sont des itérables
- le bloc d'instructions d'une boucle `for` est répété pour chacune des valeurs de la séquence
- dans ce bloc d'instructions, la *variable d'itération* est associée à la valeur de l'itération courante

On découvre

- comment manipuler des collections de valeurs
- comment écrire des listes Python
- comment parcourir une liste Python

Il est parfois nécessaire de manipuler une collection de valeurs.
Par exemple le nombre de jours de chacun des mois de l'année, ou les notes de l'ensemble des étudiants de 1ère année de licence, *etc.*

En Python, on utilisera, par exemple, les *listes*.

Comme les chaînes de caractères, les listes sont des *itérables* : on va donc pouvoir réaliser une même instruction sur chacun des éléments d'une liste.

Nous découvrirons ensuite d'autres caractéristiques des listes Python.

1 Manipuler des listes

1.1 Construire une liste via une séquence d'éléments

En Python, les *listes* sont représentées par une suite de valeurs ou d'expressions séparées par des virgules `,` et entourées de crochets `[` et `]`.

```
liste = [<expression1>, <expression2>, <expression3>, ...]
```

Une liste composée d'une seule valeur est appelée singleton. La liste vide ne contient aucun élément.

Par exemple :

```
>>> [31, 28, 31, 30]
[31, 28, 31, 30]
>>> [3 * 10, 40 // 5]
[30, 8]
>>> ["informatique", "computer science", "infor" + "matica"]
['informatique', 'computer science', 'informatica']
>>> [3 < 14, '3' < '14', 3.14 < 3.15, type(3.14) == float]
[True, False, True, True]
>>> ["unique"] # un singleton
['unique']
>>> [] # la liste vide
[]
```

On évitera de construire des listes contenant des éléments de types différents, même si Python le permet (par ex `[1, True, 'a']`).

1.2 Type

Le type est `list`, indépendamment du type des valeurs contenues dans la liste.

```
>>> type([31, 28, 31, 30])
<class 'list'>
>>> type([])
<class 'list'>
```

1.3 Opérateurs applicables aux listes

1.3.1 Concaténation et répétition

Les opérateurs habituels de concaténation `+`, et de répétition `*` que nous avons déjà vu pour le type `str` peuvent être utilisés aussi avec des valeurs de type `list`.

```
>>> [1, 2] + [3, 4]
[1, 2, 3, 4]
>>> [1, 2] * 3
[1, 2, 1, 2, 1, 2]
```

On notera que l'opérateur de concaténation s'applique à des valeurs de type `list`.

Pour créer une nouvelle liste `[1, 2, 3]` contenant les éléments d'une liste `[1, 2]` auquels on a ajouté un élément supplémentaire `3`, on n'écrit pas :

```
>>> [1, 2] + 3
[...]
TypeError: can only concatenate list (not "int") to list
```

mais on concatène à la liste `[1, 2, 3]` une liste singleton `[3]` :

```
>>> [1, 2] + [3]
[1, 2, 3]
```

On construit ci-dessous la liste contenant le nombre de jours des mois d'une année non bissextile :

```
>>> mois_reguliers = ([31, 30] * 2) + [31]
>>> mois_reguliers
[31, 30, 31, 30, 31]
>>> nb_jours_mois = [31, 28] + mois_reguliers * 2
>>> nb_jours_mois
[31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
```

1.3.2 Comparaisons

Les opérateurs d'égalité `==` et d'inégalité `!=` sont également applicables aux listes.

Deux listes sont égales si et seulement si elles contiennent exactement les même valeurs, dans le même ordre.

```
>>> [1, 2, 4] == [1, 1+1, 2*2]
True
>>> ['titi', 'toto'] == ['titi', 'toto', 'tata']
False
>>> ['First', 'Second'] == ['Second', 'First']
False
>>> ['First', 'Second'] != ['Second', 'First']
True
```

1.4 Afficher une liste

Voilà ce que produit l'affichage d'un littéral de type liste :

```
>>> print([1, 2, 3])
[1, 2, 3]
```

1.5 Quelques fonctions et méthodes liées aux listes

1.5.1 Longueur d'une liste : `len()`

Comme pour le type `str`, la fonction `len()` peut également prendre en paramètre une liste et renvoyer le nombre d'éléments qu'elle contient.

```
>>> len([1, 2, 3])
3
>>> len([])
0
```


1.5.2 Modification d'une liste par ajout d'un élément à sa fin : `list.append()`

Nous avons vu comment la concaténation de 2 listes produit une troisième liste.

La méthode `list.append()` permet de **modifier** une liste en lui ajoutant un élément à la fin (on dit aussi *insérer un élément*).

La syntaxe est la suivante :

```
<liste>.append(<elt>)
```

Par exemple :

```
>>> a_reviser = ['chaînes', 'for']
['chaînes', 'for']
>>> a_reviser.append('listes')
>>> a_reviser
['chaînes', 'for', 'listes']
```

La méthode `append` *ne renvoie rien* (dit autrement : elle renvoie `None`, ça devrait vous faire penser à la fonction `print`).

```
>>> [1, 2].append(3) # s'évalue à None, donc Python n'affiche rien
>>> l = [1, 2] # affectation, Python n'affiche rien non plus
>>> l
[1, 2]
>>> l.append(3) # l.append(3) s'évalue à None, donc Python n'affiche rien
>>> l # valeur de l après modification
[1, 2, 3]
>>> l.append(4).append(5)
[...]
AttributeError: 'NoneType' object has no attribute 'append'
```

De même une fonction qui effectuerait un `return l.append(3)` renverra toujours `None`.

Attention : l'élément ajouté en fin de liste peut être de n'importe quel type, y compris de type `list` :

```
>>> l = [1, 2]
>>> l.append([3])
>>> l
[1, 2, [3]]
```

NB : Une telle méthode de modification par ajout en fin n'existe pas sur les chaînes de caractères, nous y reviendrons.

1.5.3 Conversions

Il est possible de convertir un itérable en liste en utilisant la fonction prédéfinie `list()`.
Cette fonction crée une liste dont les éléments sont ceux de l'itérable.

Nous pouvons l'utiliser sur les itérables que nous connaissons, les chaînes de caractères.

```
>>> list("bonjour")
['b', 'o', 'n', 'j', 'o', 'u', 'r']
```

Par contre :

```
>>> str([1, 2, 3])
'[1, 2, 3]' # et non "123"
```

2 Itérer sur les éléments d'une liste

Les listes sont des *itérables*. On peut donc les parcourir à l'aide d'une boucle `for`.

À chaque itération, la variable d'itération sera associée à l'élément courant de la liste.

Avec un itérable de type liste, la variable d'itération peut prendre le nom générique `elt` ou `e` pour **e**lement (les accents étant à éviter dans les noms de variables ou fonctions). Il est courant de nommer :

- la liste selon les éléments qu'elle contient, par exemple `lprix` pour une liste de prix
- la variable d'itération par le nom d'un élément, par exemple `prix`.

2.1 Un exemple

Pour présenter un exemple de parcours de liste, nous allons calculer la somme de ses éléments.

```
somme = 0
l = [12, 30, 5]
for elt in l:
    somme = somme + elt
```

Avant l'entrée dans la boucle `for`, la somme `s` est *initialisée* à `0` (instruction d'initialisation). Lors de l'exécution de la boucle, on aura :

- lors du premier tour de boucle, `elt` prend la valeur `12`,
  et `somme` prend la valeur `12` (`0 + 12`)
- lors du deuxième tour de boucle, `elt` prend la valeur `30`,
  et `somme` prend la valeur `42` (`12 + 30`)
- lors du troisième et dernier tour de boucle, `elt` prend la valeur `5`, et `somme` prend la valeur `47` (`42 + 5`)

La valeur finale de `somme`, en sortie de boucle, sera donc `47`.

On note qu'on a initialisé `somme` avec la valeur `0`, élément neutre de l'addition. Ce qui permet d'établir que la somme des éléments d'une liste vide vaut `0`. Dans ce cas, aucune itération n'est effectuée.

```
>>> somme = 0
>>> for elt in []:
        somme = somme + elt
>>> somme
0
```

3 Memento

- les listes sont des séquences d'éléments de types quelconques
- l'écriture littérale d'une liste est une séquence d'expressions séparées par des virgules `,` et encadrée de crochets `[` et `]`
- on concatène des listes avec l'opérateur `+`
- on peut répéter une liste avec l'opérateur `*`
- on ajoute un élément à la fin d'une liste à l'aide de la méthode `append()`
- les listes sont des itérables, on les parcourt avec une boucle `for`

Python propose un type itérable qui permet de représenter la séquence des valeurs entières contenues dans un intervalle. Ce type s'appelle `range`.

Pour créer une telle séquence, on fournit :

- la borne inférieure de l'intervalle (cette borne est *incluse* dans la séquence)
- la borne supérieure de l'intervalle (cette borne est *exclue* de la séquence).

On peut ne pas fournir la borne inférieure, qui vaut alors par défaut `0`.

Par exemple :

- `range(2, 6)` contient la séquence des valeurs entières `2`, `3`, `4`, `5`
- `range(6)` contient la séquence des valeurs entières `0`, `1`, `2`, `3`, `4`, `5`

On a bien :

```
>>> type(range(6))
<class 'range'>
```

L'impression d'un objet de type `range` n'affiche pas ses éléments :

```
>>> print(range(2, 6))
range(2, 6)
```

Comme un objet de type `range` est itérable, on peut parcourir ses valeurs avec une boucle `for`.

Pour afficher les éléments d'un `range`, on pourra par exemple écrire la fonction :

```
def affiche_range(binf:int, bsup:int):
    ''' Affiche les valeurs de range(binf, bsup)

    precondition : /
    '''
    for elem in range(binf, bsup):
        print(elem)
```

On constate alors qu'on peut créer des intervalles ne contenant aucune valeur :

```
>>> affiche_range(2, 6)
2
3
4
5
>>> affiche_range(6, 2)
>>> affiche_range(6, 6)
>>>
```

On constate que `range(n)` contient `n` valeurs, pour `n` positif ou nul. Itérer sur `range(n)` exécutera donc `n` itérations.

Pour demander `n` saisies à l'utilisateur, ou encore générer une liste contenant `n` valeurs, on veut justement itérer `n` fois.

Comme on n'utilisera alors pas les valeurs contenues dans l'intervalle `range(n)`, on pourra utiliser un souligné `_`[1] à la place de la variable d'itération :

```
for _ in range(n):
   ...
```

Par exemple :

```
def saisie_caracteres(n:int) -> str:
    """ Demande la saisie de n caractères au clavier et renvoie
    la chaîne résultante.
```

[1] Du 8, pas du 6 !

```
    precondition : n >= 0
    Exemples : /
    """
    res = ''
    for _ in range(n):
        car = input('Entrer un caractere : ')
        res = res + car
    return res
```

Attention : pour créér l'intervalle `[1..n]`, il faut utiliser `range(1, n+1)`.

On peut convertir un `range` en liste (on obtient la liste de ses élements) mais on ne pourra pas obtenir une chaîne sur le même principe :

```
>>> list(range(6))
[0, 1, 2, 3, 4, 5]
>>> str(range(6))
'range(0, 6)'
```