Types construits

5.1

Principe général

5.1.1 Type Simple, principe

Le type simple fait référence à un contenu unique.

On peut voir une variable de ce type comme une **liaison** entre un **NOM** et un **CONTENU** en mémoire.

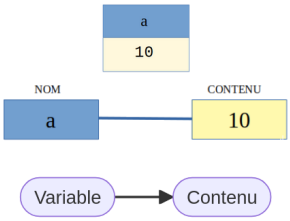

Figure 5.1 – Principe du type simple

5.1.2 Type Construit, principe

Le type construit fait plutôt référence à une armoire qui contient plusieurs boites. Ici, un tableau `t1` qui fait réference à 10, 100 et 1000.

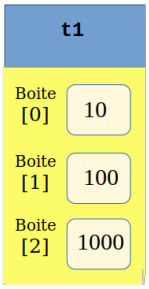

Figure 5.2 – Armoire

Il est néanmoins préférable d'y voir un NOM qui permet d'atteindre un ensemble de CONTENUS, et plus un contenu unique.

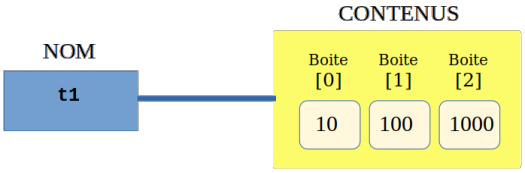

Figure 5.3 – Principe de la liaison

On obtient alors ce schéma de principe :

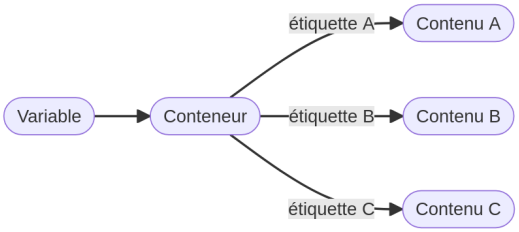

Figure 5.4 – Principe du type construit

5.2

Strings

5.2.1 Définition : voir Type simple 4.2.1

5.2.2 Déclaration : voir Type simple 4.2.2

5.2.3 Opérateurs : voir Type simple 4.2.3

5.2.4 Longueur : voir Type simple 4.2.4

Voyons maintenant les nouveautés.

5.2.5 Accéder à une case [i]

Accés

On peut accéder **à l'élément stocké dans une case dont on connait l'indice** en tapant le nom de la variable-string suivie de **crochets** et de l'indice.

```
Indice      0123456
>>> mot = "Bonjour"
```

```
>>> mot[0]
'B'
```

```
>>> mot[1]
'o'
```

```
>>> mot[2]
'n'
```

```
>>> mot[7]
IndexError: list index out of range
```

Trois choses à distinguer

— Le conteneur ⇒ **s** (le string)
— Les indices ⇒ **i**
— Les contenus ⇒ **s[i]**

5.2.6 Immuable en Python

En Python, les strings sont **immuables** (ou **non mutables** en franglais) : on ne peut **pas modifier le contenu d'une case après la création** du string.

```
>>> a = "bonjour"
>>> a[0] = "d"
TypeError: 'str' does not support item assignment
```

5.2.7 Parcours par indices

On utilise une boucle **for i in range(len(s))**.

```
1  s = "bonjour"
2
3  for i in range(len(s)) :
4      print(s[i])
```

L4 : Pour chaque indice possible de s

L5 : Affiche le contenu de la case i

Ce programme est équivalent à :

```
1  s = "bonjour"
2
3  print(s[0])
4  print(s[1])
5  print(s[2])
6  print(s[3])
7  print(s[4])
8  print(s[5])
9  print(s[6])
```

Ils affichent l'un et l'autre ceci dans la console :

```
b
o
n
```

```
j
o
u
```

```
r
```

5.2.8 mot-clés in et not in

Test d'appartenance ou de présence

L'opérateur binaire **in** qui renvoie un booléen. Pour savoir si le string **contenu** est présent dans le string **s**, on tape **contenu in s**.

```
>>> "bon" in "bonjour"
True
```

```
>>> "Bon" in "bonjour"
False
```

Test de non-appartenance ou d'absence

L'opérateur binaire **not in** qui renvoie un booléen. Pour savoir si le string **contenu** n'est pas présent dans le string **s**, on tape **contenu not in s**.

```
>>> "bon" not in "bonjour"
False

>>> "Bon" not in "bonjour"
True
```

5.3 Tableau statique

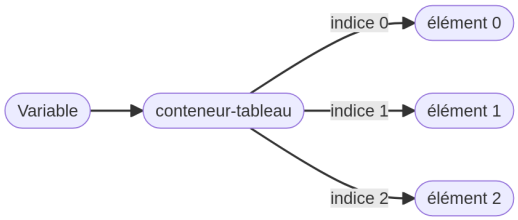

FIGURE 5.5 – Principe du tableau

5.3.1 Définition

Un **tableau statique** est un tableau comportant **un nombre de cases fixé** à la création.

Un **tableau** est un conteneur formant une **collection ordonnée d'éléments ayant tous le même type**.

On peut donc avoir un tableau d'integers, ou un tableau de floats par exemple.

Les cases sont identifiées par un numéro nommé **indice**.

Voici un exemple de tableau de caractères de 3 cases (indices 0-1-2)

Indice	0	1	2
Elément	'A'	'B'	'C'

5.3.2 Déclaration

Les tableaux statiques sont implémentés en Python sous forme d'une structure de données dont le type se nomme **list**.

A - Déclaration

On peut enregistrer une séquence de données dans un tableau statique en utilisant la syntaxe suivante :

— un crochet `[` en ouverture,
— un crochet `]` en fermeture et
— des virgules pour séparer les valeurs.

Exemple avec un tableau contenant les notes de quatre élèves :

```
>>> notes = [20, 8, 18, 12]
>>> type(notes)
<class 'list'>
```

B - Grand tableau

On peut déclarer un tableau sur plusieurs lignes en tapant sur ENTREE après chaque virgule. Exemple :

```
1  eleves = ["Lisa",
2      "Scott",
3      "Matthias",
4      "Antoine",
5      "Ethan",
6      "Lucas",
7      "Manon"]
```

Cela prend plus de lignes mais on comprend un peu mieux ce qu'on place où.

C - Tableau vide

Deux solutions

```
>>> t = []
>>> t
[]
```

```
>>> t = list()
>>> t
[]
```

5.3.3 Opérateurs Python

▷ **Concaténation** avec `+` : `list + list -> list`

```
>>> [10, 20] + [5, 30]
[10, 20, 5, 30]
```

▷ **Répétition** avec `*` :

— `list * int -> list`
— `int * list -> list`

```
>>> [10] * 3
[10, 10, 10]
```

```
>>> 3 * [12]
[12, 12, 12]
```

5.3.4 Accéder à une case [i]

Accés

On peut accéder **à l'élément stocké dans une case dont on connait l'indice** en tapant le nom de la variable-tableau suivie de **crochets** et de l'indice.

```
Indice        0  1   2   3
>>> notes = [20, 8, 18, 12]
```

```
>>> notes[0]
20

>>> notes[1]
8
```

```
>>> notes[2]
18

>>> notes[3]
12
```

```
>>> notes[4]
IndexError: list index out of range
```

Trois choses à distinguer

— Le conteneur ⇒ `t` (le tableau)
— Les indices ⇒ `i`
— Les contenus ⇒ `t[i]`

5.3.5 Déterminer sa longueur

La longueur d'un tableau correspond au nombre de "cases-éléments" du tableau.

On utilise la fonction native `len()`.

```
Indice        0  1   2   3
>>> notes = [20, 8, 18, 12]
>>> len(notes)
4
```

4 notes, des indices allant de 0 à 3.

5.3.6 Muable en Python

En Python, les tableaux sont **muables** : on peut **modifier le contenu d'une case après la création** du tableau.

```
Indice        0  1   2   3
>>> notes = [20, 8, 18, 12]
>>> notes[1] = 11
>>> notes
[20, 11, 18, 12]
```

Ce n'est pas une affectation sur le tableau lui-même. L'affectation est faite sur l'une des cases `notes[i]` du tableau.

5.3.7 Parcours par indices

On utilise une boucle `for i in range(len(t))`.

```
1  t = [20, 8, 18, 12]
2
3  for i in range(len(t)) :
4      print(t[i])
```

Traduction L3 : Pour chaque indice possible dans t

Traduction L4 : Affiche le contenu de la case i

Ce programme est équivalent à ceci :

```
1  t = [20, 8, 18, 12]
2  print(t[0])
3  print(t[1])
4  print(t[2])
5  print(t[3])
```

Ils affichent l'un et l'autre ceci dans la console :

```
20 | 8 | 18 | 12
```

5.3.8 Mot-clés in et not in

Test d'appartenance ou de présence

L'opérateur binaire **in** qui renvoie un booléen. Pour savoir si le **contenu** est présent dans le tableau **t**, on tape **contenu in t**.

```
>>> "Alice" in ["Alice, "Bob", "Charlie"]
True

>>> "Dana" in ["Alice, "Bob", "Charlie"]
False
```

Test de non-appartenance ou d'absence

L'opérateur binaire **not in** qui renvoie un booléen. Pour savoir si le **contenu** n'est pas présent dans le tableau **t**, on tape **contenu not in t**.

```
>>> "Alice" not in ["Alice, "Bob", "Charlie"]
False

>>> "Dana" not in ["Alice, "Bob", "Charlie"]
True
```

5.4 n-uplet

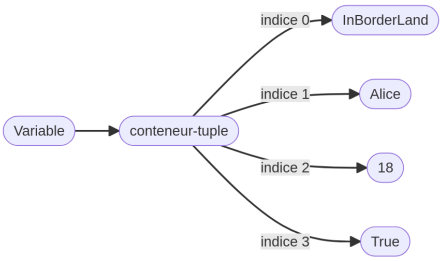

FIGURE 5.6 – Principe du tuple

5.4.1 Définition

Généralités Un **n-uplet** est un conteneur formant une **collection ordonnée d'éléments pouvant avoir un type différent les uns des autres**.

Les cases sont identifiées par un **indice**.

Un 4-uplet dont les indices sont 0-1-2-3 :

Indice	0	1	2	3
Elément	'"In Bordeland"'	"Alice"	18	True

Vocabulaire

— 1-uplet : singleton
— 2-uplet : couple
— 3-uplet : triplet
— 4-uplet : quadruplet

5.4.2 Déclaration

Les n-uplets sont implémentés en Python sous forme d'une structure de données dont le type se nomme **tuple**.

A - Déclaration

On peut enregistrer une séquence de données dans un n-uplet en utilisant la syntaxe suivante :

— une parenthèse **(** en ouverture,
— une parentèhse **)** en fermeture et
— des virgules pour séparer les valeurs.

Exemple avec un élève : nom, prénom, age, NSI ? :

```
>>> eleve = ("In Bordeland", "Alice", 18, True)
>>> eleve
('In Bordeland', 'Alice', 18, True)

>>> type(eleve)
<class 'tuple'>
```

B - Grand n-uplet

On peut déclarer un tuple sur plusieurs lignes en tapant sur ENTREE après chaque virgule. Exemple :

```
1  eleve = ("In Borderland",
2       "Alice",
3       18,
4       True)
```

C - n-uplet vide

Deux solutions

```
>>> tup = ()
>>> tup
()
```

```
>>> t = tuple()
>>> t
()
```

D - cas du singleton

Pour déclarer un tuple de 1 élément, il faut **nécessairement placer une virgule**, sinon l'interpréteur Python prendra vos parenthèses comme de simples parenthèses.

`(10,)` correct pour du tuple

`(10)` : incorrect pour du tuple

```
>>> tp1 = (10,)
>>> tp1
(10,)

>>> type(tp1)
<class 'tuple'>
```

```
>>> tp2 = (10)
>>> tp2
10

>>> type(tp2)
<class 'int'>
```

5.4.3 Opérateurs Python

▷ **Concaténation** avec `+` : `tuple + tuple -> tuple`

```
>>> (10, 20) + (5, 30)
(10, 20, 5, 30)
```

▷ **Répétition** avec `*` :

— `tuple * int -> tuple`

— `int * tuple -> tuple`

```
>>> (10,) * 3
(10, 10, 10)

>>> 3 * (10, 12)
(10, 12, 10, 12, 10, 12)
```

▷ **Erreur sémantique courante** Nous avions vu l'erreur sémantique suivante :

```
>>> 5,2 + 1,2
(5, 3, 2)
```

Pour la comprendre, il faut se rendre compte qu'on demande à Python d'interpréter un tuple ! Voici ce que comprend l'intepréteur Python :

```
>>> (5,2 + 1,2)
(5, 3, 2)
```

On lui demande donc d'évalue un triplet contenant 5 sur l'indice 0, 2 + 1 sur l'indice 1 et 2 sur l'indice 2. Moralité : explicite c'est mieux qu'implice.

5.4.4 Accéder à une case [i]

Accés

On peut accéder **à l'élément stocké dans une case dont on connait l'indice** en tapant le nom de la variable-tuple suivie de **crochets** et de l'indice.

```
Indice                0          1      2   3
>>> eleve = ("In Bordeland", "Alice", 18, True)
>>> eleve[0]
'In Borderland'
```

```
>>> eleve[1]
'Alice'
```

```
>>> eleve[2]
18
```

```
>>> eleve[3]
True
```

```
>>> notes[4]
IndexError: list index out of range
```

Trois choses à distinguer

— Le conteneur ⇒ `tup` (le n-uplet)

— Les indices ⇒ `i`

— Les contenus ⇒ `tup[i]`

5.4.5 Déterminer sa longueur

La longueur d'un n-uplet correspond au nombre n de "cases-éléments" du tableau.

On utilise la fonction native `len()`.

```
Indice                0          1      2   3
>>> eleve = ("In Bordeland", "Alice", 18, True)
>>> len(notes)
4
```

4 notes, des indices allant de 0 à 3.

5.4.6 Immuable en Python

En Python, les tuples sont **immuables** : on ne peut **pas modifier le contenu d'une case après la création** du n-uplet.

```
Indice                0          1      2   3
>>> eleve = ("In Bordeland", "Alice", 18, True)
>>> eleve[2] = 19
TypeError: 'tuple' does not support item assignment
```

5.4.7 Parcours par indices

On utilise une boucle `for i in range(len(tup))`.

```
1  eleve = ("In Borderland", "Alice", 18, True)
2
3  for i in range(len(eleve)) :
4      print(eleve[i])
```

Traduction L3 : Pour chaque indice possible dans eleve

Traduction L4 : Affiche le contenu de la case i

Ce programme est équivalent à ceci :

```
1  eleve = ("In Borderland", "Alice", 18, True)
2
3  print(eleve[0])
4  print(eleve[1])
5  print(eleve[2])
6  print(eleve[3])
```

Ils affichent l'un et l'autre ceci dans la console :

```
'In Borderland'
'Alice'
18
True
```

5.4.8 Mot-clés in et not in

Test d'appartenance ou de présence

L'opérateur binaire **`in`** qui renvoie un booléen. Pour savoir si le **`contenu`** est présent dans le n-uplet **`tup`**, on tape **`contenu in tup`**.

```
>>> "Alice" in ("Alice", "Bob", "Charlie")
True

>>> "Dana" in ("Alice", "Bob", "Charlie")
False
```

Test de non-appartenance ou d'absence

L'opérateur binaire **`not in`** qui renvoie un booléen. Pour savoir si le **`contenu`** n'est pas présent dans le n-uplet **`tup`**, on tape **`contenu not in tup`**.

```
>>> "Alice" not in ("Alice", "Bob", "Charlie")
False

>>> "Dana" not in ("Alice", "Bob", "Charlie")
True
```

5.5 Dictionnaire

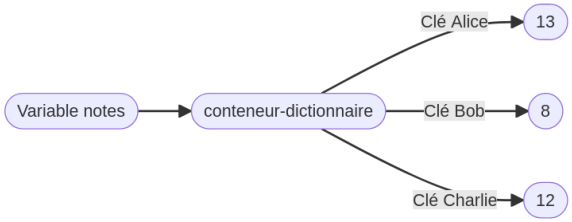

FIGURE 5.7 – Principe du dictionnaire

5.5.1 Définition d'un dictionnaire

Un dictionnaire est un **tableau associatif**. Il comporte des cases qui portent un nom et possèdent un contenu. On nomme :

- **clé (key)** le nom de la case.
- **valeur (value)** le contenu de la case.

```
Clé      "Alice"   "Bob"   "Charlie"
Valeur     13        8         12
```

5.5.2 Déclaration d'un dictionnaire

- Les éléments délimiteurs sont les accolades **`{}`**.
- On fournit un couple **(cle, valeur)** en utilisant la syntaxe Python suivante : **`cle: valeur`**.
- Les couples sont séparés par une **virgule**.

Exemple une ligne ou multiligne

```
ds = {"Alice": 13, "Bob": 8, "Charlie": 12}

ds = {
    "Alice": 13,
    "Bob": 8,
    "Charlie": 12
    }
```

5.5.3 Lecture, accéder à une valeur

On accède à une valeur en tapant le **nom du dictionnaire suivi de la clé entre crochets**.

```
>>> ds = {"Alice": 13, "Bob": 8, "Charlie": 12}
>>> ds["Bob"]
8

>>> ds["Bob l'éponge"]
KeyError: "Bob l'éponge"
```

5.5.4 Opérateurs Python

Ni concaténation, ni répétition. C'est fini.

5.5.5 Déterminer sa longueur

On peut utiliser la fonction native **`len()`** pour obtenir le nombre de couples **`(clé, valeur)`** enregistrés dans un dictionnaire.

Exemple :

```
>>> ds = {"Alice": 13, "Bob": 8, "Charlie": 12}
>>> nbr = len(ds)
>>> nbr
3
```

5.5.6 Muable en Python

En Python, les dictionnaires sont muables : on peut **modifier le contenu d'une case après la création** du dictionnaire.

Modification d'un couple existant

```
>>> {"Alice": 13, "Bob": 8, "Charlie": 12}
>>> notes["Bob"]
8

>>> notes["Bob"] = 11
>>> notes["Bob"]
11

>>> notes
{'Alice': 13, 'Bob': 11, 'Charlie': 12}
```

Notez bien que **ce n'est pas une affectation sur le dictionnaire** : l'affectation est faite sur l'un des contenus, pas sur le dictionnaire notes lui-même.

Rajout d'un nouveau couple

On peut rajouter de la même façon un couple qui n'existe pas encore. Imaginons un nouvel élève nommé David qui a eu 15.

```
>>> {"Alice": 13, "Bob": 11, "Charlie": 12}
>>> notes["David"] = 15

>>> notes
{'Alice': 13, 'Bob': 11, 'Charlie': 12, 'David': 15}
```

5.5.7 Parcours par clés

On utilise une boucle `for cle in d.keys()`.

```
1  ds = {"Alice" : 13,
2        "Bob" : 8,
3        "Charlie" : 12
4       }
5
6  for cle in ds.keys() :
7      print(ds[cle])
```

Traduction L6 : Pour chaque cle possible dans ds

Traduction L7 : Affiche la valeur associée à cette clé

Ce programme est équivalent à ceci :

```
1  ds = {"Alice" : 13,
2        "Bob" : 8,
3        "Charlie" : 12
4  }
5
6  print(ds["Alice"])
7  print(ds["Bob"])
8  print(ds["Charlie"])
```

Ils affichent l'un et l'autre ceci dans la console :

```
13 | 8 | 12
```

5.5.8 Mot-clés in et not in

Test d'existence d'une clé

L'opérateur binaire **`in`** qui renvoie un booléen. Pour savoir si la **`cle`** est une clé dans le dictionnaire **`d`**, on tape **`cle in d.keys()`** ou **`cle in d`**.

Version explicite :

```
>>> d = {"Alice": 12, "Bob": 8}
```

```
# Version explicite
>>> "Alice" in d.keys()
True

>>> "Dana" in d.keys()
False
```

```
# Version implicite
>>> "Alice" in d
True

>>> "Dana" in d
False
```

Traduction : "Alice" est-elle une clé du dictionnaire d ?

Test d'absence d'une clé

L'opérateur binaire **`not in`** qui renvoie un booléen. Pour savoir si la **`cle`** n'est pas une clé dans le dictionnaire **`d`**, on tape **`cle not in d.keys()`** ou **`cle not in d`**.

```
>>> d = {"Alice": 12, "Bob": 8}
```

```
# Version explicite
>>> "Bob" not in d.keys()
False

>>> "Da" not in d.keys()
True
```

```
# Version implicite
>>> "Bob" not in d
False

>>> "Da" not in d
True
```

Traduction : "Alice" est-elle absente en tant que clé dans le dictionnaire ?

Convertir en tableau

5.6.1 list -> list avec list()

On peut créer une copie rapide d'un tableau plutôt que d'en faire un alias vers ce même contenu-tableau.

```
>>> t = [1, 2, 4]
# création d'un tableau

>>> t2 = t
# t2 n'est qu'un alias, un autre nom

>>> t3 = list(t)
# t3 est une copie indépendante1

>>> t[0] = 100
# on modifie le contenu de la case 0 par 100

>>> t
[100, 2, 4]    # Modifié : normal

>>> t2
[100, 2, 4]    # Modifié : normal, alias de t

>>> t3
[1, 2, 4]      # Non modifié : normal, copie.
```

5.6.2 tuple -> list avec list()

Crée une copie partielle du contenu immuable qui devient mutable.

5.6.3 dict -> list avec list()

On crée un tableau contenant soit les clés, soit les valeurs, soit des tuples (clé, valeur associée).

```
>>> d = {'Alice':14, 'Bob':12}
# création d'un dictionnaire

# IMPLICITE : t contient les clés de d
>>> t1 = list(d)
>>> t1
['Alice', 'Bob']

# EXPLICITE : t contient les clés de d
>>> t2 = list(d.keys())
>>> t2
['Alice', 'Bob']

# EXPLICITE : t contient les valeurs de d
>>> t3 = list(d.values())
>>> t3
[14, 12]

# EXPLICITE : t contient les couples de d
>>> t4 = list(d.items())
>>> t3
[('Alice', 14), ('Bob', 12)]
```

5.6.4 str -> list avec list()

On récupère les caractères, un par case.

```
>>> nom = "Merlin"
>>> t = list(nom)
>>> t
['M', 'e', 'r', 'l', 'i', 'n']
```

5.6.5 str -> list avec split()

La méthode **split()** veut dire séparer et permet justement de "séparer" un string en différentes parties et placer ces parties dans un tableau.

mots d'une phrase

```
>>> nom = "Merlin ou Toto l'asticot"
>>> t = nom.split(' ')

>>> t
['Merlin', 'ou', 'Toto', "l'asticot"]
```

split sans rien

Par défaut, il est possible de ne pas préciser le caractère qui servira à scinder le string en plusieurs parties : la méthode est configurée pour considérer qu'il s'agit de l'espace.

```
>>> nom = "Merlin ou Toto l'asticot"
>>> t = nom.split()

>>> t
['Merlin', 'ou', 'Toto', "l'asticot"]
```

Récupérer les phrases ?

Pour récupérer les phrases, on peut considérer qu'il suffit d'utiliser la méthode en considérant que le caractère séparateur est le point.

Convertir en tuple

On peut utiliser **tupe()** : même principe que la partie précédente mais on crée un tuple pas un tableau.

5.8

Convertir en dictionnaire

5.8.1 dict -> dict avec dict()

Utiliser `dict()` sur un autre dictionnaire permet d'en faire une copie peu profonde.

```
# création d'un dictonnaire
>>> {'Alice': 14, 'Bob': 12}

# d2 n'est qu'un alias !
>>> d2 = d

# d3 est une copie indépendante1
>>> d3 = dict(t)

# on modifie le contenu de la case 'Alice'
>>> d['Alice'] = 20

# Modifié : normal
>>> d
{'Alice': 20, 'Bob': 12}

# Modifié : normal, c'est un alias de d !
>>> d2
{'Alice': 20, 'Bob': 12}

# Non modifié : normal, c'est une copie.
>>> d3
{'Alice': 14, 'Bob': 12}
```

5.8.2 list|tuple|str -> dict avec dict()

On ne peut pas convertir n'importe quel type de list, tuple ou str en dictionnaire.

Nous verrons plus tard dans l'année qu'on peut convertir un type particulier de tableau en dictionnaire : on peut convertir un tableau de couple en dictionnaire.

5.9

Convertir en str

5.9.1 list|tuple|dict -> str avec str()

On peut convertir n'importe quel type de list, tuple ou dict en string. Par contre, l'utilité directe est limitée. Cela peut servir pour sauvegarder un tel contenu via un fichier-texte.

```
>>> str( [14, 12] )
"[14, 12]"

>>> str( ('Alice', 12) )
"('Alice', 12)"

>>> str( {'Alice': 14, 'Bob': 12} )
"{'Alice': 14, 'Bob': 12}"
```

5.9.2 Créer un str avec join()

On peut convertir en str des cas particuliers d'autres types construits :

— Des tableaux list dont les éléments sont des str ;
— Des n-uplets tuple dont les éléments sont des str ;
— Des dictionnaires dict dont les clés sont des str.

```
>>> t = ["Ahah", "Bonbon", "Coucou"]

>>> s = "---".join(t)
>>> s
'Ahah---Bonbon---Coucou'

>>> s = ";".join(t)
>>> s
'Ahah;Bonbon;Coucou'


>>> d = {"Alice":14, "Bob":12, "Charlie":8}

>>> s = "---".join(d)
>>> s
'Alice---Bob---Charlie'

>>> s = " ".join(d)
>>> s
'Alice Bob Charlie'
```