CHAPITRE 4
LES TYPES DE BASE – PARTIE 2

Programmation en Python

Les chaînes de caractères

- Les chaînes de caractères, « strings » en anglais, désignent tout ce qui est considéré comme du texte.
- En Python, les chaînes de caractères sont des objets de type `string`.
- Les chaînes de caractères doivent toujours être entre guillemets (simples, doubles, ou triples).

```
>>> s = "Hello"
>>> type(s)
<class 'str'>
>>>
```

Les chaînes de caractères

- L'opérateur de concaténation sur les chaînes, c'est-à-dire le fait d'appondre une châine à la suite de l'autre, se note « + ».
- L'opérateur de multiplication sur les chaînes, c'est-à-dire le fait d'appondre n fois la même châine, se note « * ».

Les chaînes de caractères


```
>>> a = "Bonnie"
>>> b = "Clyde"
>>> a + " & " + b
'Bonnie & Clyde'
>>> a * 3
'BonnieBonnieBonnie'
>>>
>>> x = "chaine entre doubles guillemets"
>>> y = 'chaine entre simples guillemets'
>>> z = '''chaine entre triples guillemets'''
```

Les chaînes de caractères

- Indexation (indexing) : les caractères qui composent une chaîne de caractères s'énumèrent :
  - de gauche à droite depuis l'indice 0 jusqu'au dernier indice.
  - de droite à gauche depuis l'indice -1 jusqu'au premier indice.
- Exemple :

```
s = H e l l o
```

Indices depuis le début : 0 1 2 3 4

Indices depuis la fin : -5 -4 -3 -2 -1

Les chaînes de caractères

- Le n+1ème caractère depuis le début de la chaîne `s` s'obtient par la syntaxe `s[n]`.
- Le nème caractère depuis la fin de la chaîne `s` s'obtient par la syntaxe `s[-n]`.

Les chaînes de caractères


```
>>> a = "bonjour"
>>> a[0]
'b'
>>> a[1]
'o'
>>> a[2]
'n'
>>> a[6]
'r'
>>> a[-1]
'r'
>>> a[-2]
'u'
```

Les chaînes de caractères

- Découpage (slicing) : on peut extraire la suite de caractères qui composent une sous-chaînes d'une chaîne donnée.
- La sous-chaîne qui va du $m^{ème}$ caractère compris au $n^{ème}$ caractère non compris de la chaîne `s` s'obtient par la syntaxe `s[m:n]`.
- La longueur d'une liste est donnée par la fonction `len()`.

Les chaînes de caractères


```
>>> a = "bonjour"
>>> a[0:2]
'bo'
>>> a[2:7]
'njour'
>>> a[3:4]
'j'
>>> len(a)
7
>>>
```

Les chaînes de caractères

- En Python, les chaînes (contrairement aux listes) sont des objets de type **non-modifiable** (ou immuable).
- C'est-à-dire qu'il n'est pas possible de modifier un caractère ou une sous chaîne d'une chaîne donnée via une instruction de réaffectation.

```
>>> a = "bonjour"
>>> a[3] = "k"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Les chaînes de caractères

- Il existe plusieurs méthodes que l'on peut appliquer sur les chaînes de caractères.
- Pour appliquer une méthode à une chaîne de caractères, ou utilise la syntaxe :

```
chaine.methode(args)
```

- Pour connaître toutes les méthodes des chaînes de caractères, taper « help(str) »

Les chaînes de caractères

- `s.find(sub)`

Retourne l'indice de la sous-chaîne sub dans la chaîne s.
Retourne -1 si sub n'est pas dans s.

- `s.isalnum()`

Retourne True si tous les caractères de s sont alpha-numériques et que s possède au moins un caractère.

Retourne False sinon.

- `s.isalpha()`

Retourne True si tous les caractères de s sont alpa-bétiques et que s possède au moins un caractère.

Retourne False sinon.

Les chaînes de caractères

- `s.isdigit()`

Retourne True si tous les caractères de s sont numériques et que s a au moins un caractère.

Retourne False sinon.

- `s.islower()`

Retourne True si tous les caractères de s sont minuscules et que s a au moins un caractère.

Retourne False sinon.

- `s.isupper()`

Retourne True si tous les caractères de s sont majuscules et que s a au moins un caractère.

Retourne False sinon.

Les chaînes de caractères

- `s.lower()` `s.upper()`

Convertit s en minuscules ou en majuscules.

- `s.replace(old, new [,nb])`

Remplace dans s les sous-chaînes old par les sous-chaînes new. Le paramètre facultatif nb permet de préciser le nombre de remplacements.

- `s.strip([space])`

Supprime les blancs du début et de la fin de s.

Le paramètre space permet de préciser d'autres caractères que les blancs.

Les chaînes de caractères


```
>>> s = "TIGRE"
>>> s.lower()
tigre
>>> s
TIGRE                  # donc s n'est pas modifiée!
>>> t = s.lower() # t nouvelle chaîne créée à partir de s
>>> t
tigre
>>> s.replace("G","T")
TITRE
>>> s       # s n'est pas modifiée!
TIGRE
```

Les chaînes de caractères

- Parcours de chaînes: on peut parcourir les caractères successifs d'une chaîne via une boucle « for » (cf. chapitre suivant).
- Pour cela, ou utilise la syntaxe :

```
for var in string:
    instructions...
```

- Ne pas oublier les deux points à la fin de l'instruction « for… » et l'indentation obligatoire des instructions imbriquées dans la boucle « for ».

Les chaînes de caractères


```
>>> s = "TIGRE"
>>> for c in s:
>>> ...    print(c)
>>> ...
T
I
G
R
E
```

Programme (faire démo)


```
phrase = input("Entrer une phrase: ")
phrase = phrase.lower() # convertit en minuscules
phrase = phrase.replace(" ", "") # enlève les espaces
palindrome = True   # var bool qui donne la réponse...
g = 0               # indice 1er caractère de phrase
d = len(phrase) - 1 # indice dernier caractère de phrase
while g < d:
    if phrase[g] != phrase[d]:
        palindrome = False
        break
    else:
            g = g + 1
            d = d - 1
```

Programme (faire démo)


```
if palindrome == True:
        print("C'est un palindrome")
else:
        print("Ce n'est pas un palindrome")
```

Les listes

- Les listes sont des collections finies ordonnées d'objets. Ces objets peuvent être de types différents.
- En Python, les listes sont des objets de type `list`.
- Les syntaxe d'une liste est la suivante : les éléments sont entre crochets et séparés par des virgules.

```
>>> l = [1,2,3.5,"bonjour",True]
>>> type(l)
<class 'list'>
```

Les listes

- Comme pour les chaînes de caractères, l'opérateur de concaténation sur les listes se note « + ».
- L'opérateur de multiplication sur les listes se note « * ».
- L'indexation (indexing) et le découpage (slicing) des listes se comportent également de la même manière que pour les chaînes de caractères.

Les listes

- Le n+1ème élément depuis le début de la liste `l` s'obtient par la syntaxe `l[n]`.
- Le nème élément depuis la fin de la liste `l` s'obtient par la syntaxe `l[-n]`.
- La sous-liste qui va du mème élément compris au nème élément non compris de la liste `l` s'obtient par la syntaxe `l[m:n]`.
- La longueur d'une liste est donnée par la fonction `len()`.

Les listes


```
>>> fruits = ['orange', 'banane', 'pomme', 'cerise']
>>> len(fruits)
4
>>> fruits[0]
'orange'
>>> fruits[-1]
'cerise'
>>> fruits[-2]
'pomme'
>>> fruits[1:3]
['banane', 'pomme']
>>>
```

Les listes

- En Python, les listes (contrairement aux chaînes) sont des objets de type **modifiable**.
- C'est-à-dire qu'il est possible de modifier un élément ou une sous-liste d'une liste donnée via une instruction de réaffectation.
- Cette propriété est beaucoup utilisée en Python.

Les listes


```
>>> l = ["jaune", "orange", "rouge", "vert", "bleu"]
>>> l[0] = "jaune pale"
>>> l
['jaune pale', 'orange', 'rouge', 'vert', 'bleu']
>>> l[-1] = "bleu fonce"
>>> l
['jaune pale', 'orange', 'rouge', 'vert', 'bleu fonce']
>>> l[2:4] = ["rouge vif","vert clair"]
>>> l
['jaune pale', 'orange', 'rouge vif', 'vert clair', 'bleu
fonce']
```

Les listes

- **Attention :** dans le cas des objets de type modifiable, une copie d'un object original – obtenue via une instruction d'affectation – subira les mêmes modifications que l'objet original.

Les listes


```
>>> l1 = ["jaune", "orange", "rouge"]
>>> l2 = l1  # l2 est une copie de la liste l1
>>> l1[0] = "jaune pale"  # modification de l1
>>> l1
['jaune pale', 'orange', 'rouge'] # l1 est modifiée (normal)
>>> l2
['jaune pale', 'orange', 'rouge'] # l2 est aussi modifiée !
```

Les listes

- Si besoin, on peut tout de même créer une « copie non-modifiable » d'une liste via l'instruction de slicing suivante

```
l_copie = l[:]
```

Les listes


```
>>> l1 = ["jaune", "orange", "rouge"]
>>> l2 = l1[:]                  # l2 copie non-modifiable de l1
>>> l1[0] = "jaune pale"        # modification de l1
>>> l1
['jaune pale', 'orange', 'rouge'] # l1 est modifiée (normal)
>>> l2
['jaune', 'orange', 'rouge']      # l2 n'est pas modifiée !
```

Les listes

- Il existe plusieurs méthodes que l'on peut appliquer sur les listes.
- Pour appliquer une méthode à une liste, ou utilise la syntaxe :

```
liste.methode(args)
```

- Pour connaître toutes les méthodes des liste, taper « help(list) »

Les listes

- `l.append(x)`

Ajoute l'élément x à la fin de la liste l.

- `l.count(x)`

Retourne le nombre d'occurrences de l'élément x dans la liste l.

- `l.extend(u)`

Si u est une liste, ajoute tous les éléments de la liste u, un à un, à la fin de la liste l.

Si u est une chaîne, ajoute tous les caractères de la chaîne u, un à un, à la fin de la liste l.

Les listes

- `l.index(x)`

Retourne le premier indice de l dont la valeur est x.

Retourne ValueError si cet indice n'existe pas.

- `l.insert(index,object)`

Insère dans la liste l l'objet object juste avant l'indice index.

- `l.pop(index)`

Supprime et retourne l'objet d'indice index de l.

- `l.remove(x)`

Supprime dans l la première occurrence de l'objet x.

Les listes

- `l.reverse()`

Inverse l'ordre des éléments de la liste l.

- `l.sort()`

Si cela est possible, trie la liste l. Si les objets sont de types différents et qu'on ne peut pas les trier, revoie TypeError.

Les listes


```
>>> l = ["jaune","orange","rouge"]
>>> l.append(2.83)
>>> l
['jaune', 'orange', 'rouge', 2.83]
>>> l.count("rouge")
1
>>> l.extend([1,2,3])
>>> l
['jaune', 'orange', 'rouge', 2.83, 1, 2, 3]
>>> l.index("rouge")
2
>>>
```

Les listes


```
>>> l.insert(2,True)
>>> l
['jaune', 'orange', True, 'rouge', 2.83, 1, 2, 3]
>>> l.pop(3)
>>> l
['jaune', 'orange', True, 2.83, 1, 2, 3]
>>> l.remove("jaune")
>>> l
['orange', True, 2.83, 1, 2, 3]
>>> l.reverse()
>>> l
[3, 2, 1, 2.83, True, 'orange']
```

Les listes

- Parcours de listes : on peut parcourir les éléments successifs d'une liste via une boucle « for » (cf. chapitre suivant).
- Pour cela, ou utilise la syntaxe :

```
for var in liste:
    instructions...
```

- Ne pas oublier les deux points à la fin de l'instruction « for… » et l'indentation obligatoire des instructions imbriquées dans la boucle « for ».

Les listes

Programme

```
l = [1,2,"monday",3,4,5,"tuesday",6,7,8,9,10]
for x in l:
    if isinstance(x, str): # vérifie si x est une chaîne
        print(x)
    else:
        pass # instruction vide
```

Exécution

```
monday
tuesday
```

Les listes

- Les éléments des listes peuvent eux-mêmes être des listes. Ceci donne lieu à des listes de listes, des listes de listes de listes, etc.
- Les vecteurs sont généralement représentés par des listes et les matrices par des listes de listes.

Les listes


```
>>> l = [1,2,3,[4,5,6]]
>>> l[3]
[4,5,6]
>>> l[3][0]
4
>>> l[3][1]
5
>>> l[3][2]
6
>>>
```

Les listes


```
>>> v1 = [2.34, 4.35, 6.18]
>>> v2 = [-3.74, 7.39, 18.96]
>>> v3 = [2.34, 12.83, -26.47]
>>> m = [v1, v2, v3]
>>> m
[[2.34, 4.35, 6.18], [-3.74, 7.39, 18.96],
                     [2.34, 12.83, -26.47]]
>>> m[1][2]
18.96
```

Les listes

- La fonction `range(start,stop,[step])`, très utilisée en Python, produit un objet qui retourne une liste de valeurs telle que :
  - la première valeur est start
  - les valeurs suivantes sont incrémentées par pas de step
  - et ce jusqu'à atteindre la valeur stop non incluse
- On peut alors parcourir cet objet via la syntaxe :

```
for var in range(start,stop[,step]):
    instructions...
```

Les listes

Programme

```
for i in range(0,1000):
    if i%27 == 0:
        print(i)
```

Exécution (retourne tous les multiples de 27 entre 0 et 999 sur une colonne)

```
0
27
54
81
108
135
...
```

Programme (faire démo)


```
from random import * # importation de la librairie random
valeur = [2, 3, 4, 5, 6, 7, 8, 9, 10,"valet","dame","roi","as"]
couleur = ["pique", "coeur", "carreaux", "trefle"]
entree = ""
while entree != "q":
    entree = input("Tapez x pour tirer une carte, \
                    q pour terminer: ")
    if entree == "x":
        v = randint(0,12) # tirer une valeur au hasard
        c = randint(0,3)  # tirer une couleur au hasard
        print(str(valeur[v]) + " de " + couleur[c])
    elif entree == "q":
       print("Tirage terminé...")
    else:
        print("Erreur...")
```

Les tuples

- Les tuples sont des sortes de listes, mais de type **non-modifiable**.
- En Python, les tuples sont des objets de type `tuple`.
- Les syntaxe d'un tuple est la suivante : les éléments sont entre parenthèses et séparés par des virgules.

```
>>> t = (1,2,3.538)
>>> type(t)
<class 'tuple'>
```

Les tuples

- Comme pour les listes, l'opérateur de concaténation sur les tuples se note « + » et l'opérateur de multiplication sur les tuples se note « * ».
- L'indexation (indexing) et le découpage (slicing) des tuples se comportent également de la même manière que pour les listes.
- La longueur d'un tuple est donnée par la fonction `len()`.

Les tuples

- La grande différence est que les tuples ne possèdent pas toutes ces méthodes que l'on a sur les listes, car ce sont des objets non modifiables !
- On ne peut donc pas modifier, ajouter, supprimer, etc. d'éléments dans un tuple. Une fois celui-ci déclaré, il ne changera plus tout au long de l'exécution de votre programme.
- Par contre, on peut concaténer, multiplier, etc. des tuples pour en créer de nouveaux.

Les tuples

- Parcours de tuples : on peut parcourir les éléments successifs d'un tuple via une boucle « for » (cf. chapitre suivant).
- Pour cela, ou utilise la syntaxe :

```
for var in tuple:
    instructions...
```

- Ne pas oublier les deux points à la fin de l'instruction « for… » et l'indentation obligatoire des instructions imbriquées dans la boucle « for ».

Les tuples


```
>>> t1 = (1,2,3)
>>> t2 = (4,5,6)
>>> t1 + t2
(1,2,3,4,5,6)
>>> t1[2] = "bonjour"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Les tuples


```
>>> id = [("James","Dean"), "Université P2", "M1"]
>>> id[1] = "Université Paris 6"
>>> id[2] = "M2"
>>> id
[('James', 'Dean'), 'Université Paris 6', 'M2']
>>> id[0] = ("James","Bond")
>>> id
[('James', 'Bond'), 'Université Paris 6', 'M2']
```

- Dans ce cas, il peut être intéressant d'avoir une structure de donnée non modifiable pour stocker nom et prénom…

Les dictionnaires

- Les dictionnaires sont des tables non ordonnées de couples « clé-valeur ». Ils permettent de stocker des données, les valeurs, en fonction de certains noms qu'on leur donne, les clés.
- Par opposition aux listes, où les éléments sont indexés par des indices (nombres entiers), dans les dictionnaires, les éléments sont indexés par des clés (sortes de noms qu'on leur donne).

Les dictionnaires

- Les dictionnaires sont des structures de données de type **modifiable**.
- Les clés des dictionnaires sont des objets de types non-modifiables, comme des chaînes, des nombres, ou des tuples de chaînes et de nombres.
- Les valeurs associées aux clés peuvent être des objets de n'importe quel type (nombres, booléens, chaînes, listes, tuples, et même dictionnaires…).

Les dictionnaires

- En Python, les dictionnaires sont des objets de type `dict`.
- Les syntaxe d'un dictionnaire est la suivante : la suite des couples « clé-valeur » est entre accolades, chaque couple « clé-valeur » s'écrit sous la forme « clé : valeur », et ces couples sont sont séparés par des virgules.

```
>>> d = {"Alice": 4328, "Bob": 4357, "Celia": 4392}
>>> type(d)
<class 'dict'>
```

Les dictionnaires

- Les dictionnaires étant de type modifiable, il est possible de modifier, ajouter, supprimer des couples « clé-valeur » dans un dictionnaire via des instructions de réaffectation.
- Les principales opéations sur les dictionnaires sont :
  - obtention de la valeur associée à une clé
  - ajouter un couple « clé-valeur »
  - supprimer un couple « clé-valeur »
  - modifier un couple « clé-valeur »

Les dictionnaires

- L'obtention de la valeur associée à une clé se fait via la syntaxe :

```
dico[key]
```

- Pour l'ajout d'un couple « clé-valeur » :

```
dico[new_key] = new_value
```

- Pour la suppression d'un couple « clé-valeur » :

```
del dico[key]
```

- Pour La modification d'un couple « clé-valeur » :

```
dico[key] = new_value
```

Les dictionnaires


```
>>> d = {"Alice": 4328, "Bob": 4357, "Carmen": 4392}
>>> d
{'Carmen': 4392, 'Alice': 4328, 'Bob': 4357}
>>> d["Bob"]          # obtention de valeur associée à Bob
4357
>>> d["Dan"] = 4367   # ajout de « clé-valeur »
>>> d
{'Carmen': 4392, 'Alice': 4328, 'Dan': 4367, 'Bob': 4357}
>>> del d["Dan"]      # suppression de « clé-valeur »
>>> d
{'Carmen': 4392, 'Alice': 4328, 'Bob': 4357}
>>> d["Alice"] = 1234 # modification de « clé-valeur »
>>> d
{'Carmen': 4392, 'Alice': 1234, 'Bob': 4357}
```

Les dictionnaires

- Pour tester l'appartenance d'un couple « clé-valeur » à un dictionnaire, on utilise la syntaxe :

```
key in dico
```

```
>>> d = {"Alice": 4328, "Bob": 4357, "Carmen": 4392}
>>> "Alice" in d
True
>>> "Anna" in d
False
>>> "Anna" not in d
True
```

Les dictionnaires

- **Attention** (même remarque que dans le cas des listes) : dans le cas des objets de type modifiable, une copie d'un object original – obtenue via une instruction d'affectation – subira les mêmes modifications que l'objet original.

Les dictionnaires


```
>>> d1 = {"Alice": 4328, "Bob": 4357, "Carmen": 4392}
>>> d2 = d1                    # d2 est une copie de d1
>>> d1["Bob"] = 1111           # modification de d1
>>> del d1["Carmen"]           # modification de d1 (bis)
>>> d1
{'Bob': 1111, 'Alice': 4328}      # d1 modifié (normal)
>>> d2
{'Bob': 1111, 'Alice': 4328}      # d2 aussi modifié !
```

Les dictionnaires

- Il existe plusieurs méthodes que l'on peut appliquer sur les dictionnaires.
- Pour appliquer une méthode à un dictionnaire, ou utilise la syntaxe :

```
dico.methode(args)
```

- Pour connaître toutes les méthodes des dictionnaires, taper « help(dict) »

Les dictionnaires

- `d.clear()`

Supprime tous les couples « clé-valeur » de d.

- `d.copy()`

Retourne une copie de d.

- `d.get(key)`

Retourne la valeur associée à la clé key si elle existe.

Retourne None sinon.

Les dictionnaires

- `d.keys()`

Retourne un objet correspondant à l'ensemble des clés du dictionnaire d.

- `d.values()`

Retourne un objet correspondant à l'ensemble des valeurs du dictionnaire d.

Les dictionnaires


```
>>> d = {"Alice": 4328, "Bob": 4357, "Carmen": 4392}
>>> d.get("Alice")
4328
>>> d.keys()
dict_keys(['Bob', 'Alice', 'Carmen'])
>>> d.values()
dict_values([4357, 4328, 4392])
>>> d2 = d.copy()
>>> d2
{'Bob': 4357, 'Alice': 4328, 'Carmen': 4392}
>>> d.clear()
>>> d
{}
```

Les dictionnaires

- Parcours de dictionnaires : on peut parcourir les valeurs successives d'un dictionnaire via une boucle « for » sur les clés de celui-ci (cf. chapitre suivant).
- Pour cela, ou utilise la syntaxe :

```
for var in dico.keys():
    instructions...
```

- Ne pas oublier les deux points à la fin de l'instruction « for… » et l'indentation obligatoire des instructions imbriquées dans la boucle « for ».

Les dictionnaires


```
>>> my_dico = {"Person1": ["Alice",2308,"M1"], \
               "Person2": ["Bob",3407,"L3"], \
               "Person3": ["Carmen", 3276,"M2"]}
>>> for k in my_dico.keys():
        print(k, my_dico[k])
Person3 ['Carmen', 3276, 'M2']
Person2 ['Bob', 3407, 'L3']
Person1 ['Alice', 2308, 'M1']
```

Programme (faire démo)


```
carnet = {} # création du dictionnaire

print("n : Introduire une nouvelle adresse")
print("r : Rechercher une adresse")
print("f : Fin")

while True:
# cette boucle est exécutée tout le temps, i.e.,
# tant qu'elle n'est pas interrompue par break
```

Programme (faire démo)


```
action = input("Que voulez-vous faire? ")
if (action in ["n", "N"]):
    # introduire une nouvelle adresse
    nom = input("Entrer un nom: ")
    if nom in carnet.keys():
        print("Ce nom existe deja...")
    else:
        anniv = input("anniversaire <j.m.a>: ")
        tel = input("téléphone: ")
        carnet[nom] = (anniv, tel)
```

Programme (faire démo)


```
elif (action in ["r", "R"]):
    nom = input("Entrer un nom: ")
    if nom in carnet.keys():
        print("anniversaire:", carnet[nom][0])
        print("telephone:", carnet[nom][1])
    else:
        print("Ce nom n'existe pas...")
elif action in ["f", "F"]:
   # impression du dico carnet
   for k in carnet.keys():
         print(k, carnet[k][0], carnet[k][1])
   break # interruption de la boucle while
```

Transtypage

- Transtypage : les fonctions `int(arg)`, `float(arg)` et `str(arg)` permettent (dans la limite du possible) de convertir leur arguement en un entier, un flottant et une chaîne, respectivement.
- Ces fonctions de transtypage sont couramment utilisées.

Transtypage


```
>>> int("2")
2
>>> float("2.34")
2.34
>>> str(18)
'18'
>>> str(7.56)
'7.56'
>>> str([1,2,3])
'[1, 2, 3]'
```