UNIVERSITÉ TOULOUSE III PAUL SABATIER

Université de Toulouse

Faculté des Sciences et d'Ingénierie

DIU NSI

Bloc 1

Programmation python

Introduction, python basique et premiers types complexes

Armelle Bonenfant

Juin 2019

Introduction

- Qui suis-je ? programmeur(se) et enseignant(e) expérimenté(e) en informatique auprès d'élèves du supérieur
- Qui êtes-vous ? déjà programmeur(se), même peu, dans un langage impératif. Enseignant(s) expérimenté(e) (pas forcément en informatique) auprès d'élèves de lycée.
- Où allons-nous ?
  - apprentissage (ou révision) langage python
  - types et algorithmes simples à moyennement complexes
  - ponctué de qq astuces/techniques/retours d'expériences

Introduction

- Différents langages et python
- Enseigner
- Vocabulaire

Python basique

- Variables, (non) typage et entrées/sorties
- Structures de contrôle
- Fonctions

Premiers types complexes

- Listes
- Mutabilité
- Tuples

Introduction - Différents langages et python

Extraits Bruno Mermet — Université Le Havre

Trois principaux paradigmes[1] de programmation :

- Programmation impérative (Java, C, Python, Ada, Javascript...)
- Fonctionnelle (ML, Ocaml, Haskell...)
- Logique (Prolog)

1. c'est-à-dire façon d'envisager l'exécution

Paradigmes secondaires

- Programmation orientée objet
- Programmation évènementielle
- Programmation orientée aspect

Exécution

Selon les langages, le code source peut être interprété directement lors de l'exécution, ou bien ce peut être une traduction de ce code source dans un autre langage qui est exécutée.

- Interprété (Python, Javascript, PHP...)
- Compilé (C, Pascal...)
- Semi-compilé (Java)
- Just In Time (Java, Python)

Pourquoi python à l'UT3 ?

- De plus en plus répandu (milieux académiques et industriels)
- Prêt à l'usage (installation minimale)
- Prêt à l'usage (petite marche d'entrée)
- Public plus large qu'informaticien (Physiciens, Biologistes...)
- ...

Pièges et particularités

- L'INDENTATION ! ! ! (selon outils espace et pas tab)
- Légères différences avec python 2 : raw_input, print, // (division entière)...
- Absence de typage (inconsistence de typage)
- Beaucoup de raccourcis syntaxiques à complexité inconnue...

Quel support pour enseigner ?

- Du langage algorithmique (français "si alors sinon") ?
  - On introduit un langage qui n'est pas "exécutable", qui ne peut pas être rigoureux, qui rajoute "une couche", un intermédiaire pas forcément nécessaire
- Des shémas (algorigrammes, framework) ?
  - On introduit un formalisme rigoureux, pas toujours naturel. Cela aide certains à la représentation.
- Directement dans le langage destination (python) ?
  - Nécessite d'y associer la pratique en même temps.

Quelques outils - logiciels

- "Rien" : la plupart des systèmes (linux) ont un python sur console/terminal
- IDLE, facile à installer... (un peu trop simpliste ?)
- spyder (et la suite anaconda), plus compliqué à installer, gestion de projet, debugger
- pycharm (version académique gratuite), un peu compliqué à installer, gestion de projet, debugger
- jupyter notebook (avec anaconda ou pycharm), mélange texte/fenêtre d'interprétation, pratique pour enseigner (CTD-TD)
- ... en ligne repl.it très complet
- ... en ligne http://pythontutor.com pédagogique pour "suivre" la mémoire/les variables
- ... en ligne https://edupython.tuxfamily.org
- ... en ligne http://pixal.univ-tlse3.fr déployé pour l'UT3

Introduction - Vocabulaire

3 points de vue

- Programme
- Mémoire (UC)
- Ecran

TODO : rajouter schéma

Introduction - Vocabulaire

3 "acteurs"

- Programmeur
- Utilisateur
- Ordinateur

Introduction - Vocabulaire

"reçoit" vs "égal", "égal" vs "égal égal"

```
1 x = 5
```

n'a pas le même sens (en python) que

```
1 x == 5
```

- bien formaliser que ce n'est pas symétrique

"paramètre" vs "argument"

```
1 def nom_fonction(a,b) : # paramètres
2     ... corps de la fonction
3
4 ...
5 x = nom_fonction(x,y) # arguments
```

- différencier les paramètres qui sont dans la définition, des arguments qui sont utilisés à l'appel

"définition" de fonction et "appel" de fonction

```
1 # DEFINITION
2 def nom_fonction(a,b) :
3     ... corps de la fonction
4
5 # APPEL
6 x = nom_fonction(x,y)
```

- bien identifier que la définition est une façon de stocker "pour plus tard" l'aspect référence de l'appel

erreur de "compilation" (interprétation) vs erreur d'exécution vs conception

```
1 if x!= 0 # oubli du ":"
2    print(z/x)
```

et

```
1 print(z/x) # si x est nul
```

ne provoquent pas la même erreur ("invalid syntax" vs "division by zero")

- différencier les types d'erreur pour mieux corriger

Introduction - Vocabulaire

liste (python), liste (C = liste chaînée), tableau...

	signification du "insert"
liste	pas de complexité ajoutée
liste chaînée	parcours pour trouver la bonne place d'insertion
tableau	on agrandit le tableau et on décale tout...

- Les contraintes associées à ces types sont implicites pour les "habitués", pas pour les élèves

Dernière recommandation ...

L'ordinateur est bête... il ne ment jamais... Ne pas en douter !

Python basique

Python basique

- Variables, (non) typage et entrées/sorties
- Structures de contrôle
- Fonctions

Qu'est-ce qu'une variable ?

```
1 x = 5
2 y = "abc"
3 z = 5 + 12
```

- un identifiant, le nom de la variable. CHOIX PRIMORDIAL. Eviter les accents (encodage différent selon outils, systèmes...)
- un contenant, la case mémoire "physique"
- un contenu, la valeur
- un type, le domaine. En résumé les opérations qu'on peut utiliser sur cette variable

Analogie avec des boîtes aux lettres (destinataire, boîte aux lettres, courrier, genre de courrier que peut contenir la boîte). Ou pots de peinture (nom de la couleur, pot de peinture, quantité de peinture restante et couleur, genre de peinture que peut contenir le pot).

Particularité en python

- Pas de déclaration
- Typage inféré (deviné par python)
- Pas de persistence de type

```
1 x = 5
2 print(type(x)) # va repondre int
3 x = "abc" # aucun probleme
4 print(type(x)) # va repondre str
```

Types scalaires standards

- int
- float (opérations compatibles avec int, conversion vers le plus général si besoin)
- bool
- str

```
1 x = 3
2 y = 22.4
3 x = x + y
4 print(x)
5 # affiche 25.4
```

Lien vers tableau des opérateurs et priorités python

Opérations entières à noter

- `//` pour la division entière
- `%` pour le modulo

Bizarreries python

Quand c'est "interprétable", python interprète...

```
1 mot = "a" *3 # mot reçoit "aaa"
```

Affectation multiple

```
1 x, y = 4, "abc" # deux affectations simultanées
2 x, y = y , x # échange de variables
```

Extrêmement pratique et utilisé (notamment pour les retours de fonctions).

Entrées

```
1 x=input() # minimaliste , toujours str
2
3 x = int(input()) # conversion en int
4 x = float(input()) # conversion en float
5
6 x = int(input("saisissez un entier")) # avec message "intégré"
```

Sorties

```
1 x = "123"
2 print(x) # affiche 123 puis saut de ligne
3 y = "45"
4 print(x,y) # affiche 123 45 puis saut de ligne
5 print(x,y,sep="...") # affiche 123...45 puis saut de ligne
6 print("abc" + x) # affiche abc123 puis saut de ligne
7 print("123", end="") # pas de saut de ligne
```

Démonstration - Exercice

Voir Notebook dans le dossier...

TODO : rajouter au notebook/ou cahier d'exercices des expressions à évaluer

3 structures de contrôle

- Séquence
- Sélection
- Répétition

Et ça suffit pour TOUT programmer.... ! ! !

Séquence

Une suite finie d'actions ($action_1$, $action_2$, ..., $action_n$) qui s'exécutent d'une façon inconditionnelle.
La syntaxe (PAS DE ";") :

```
1 action 1
2 action 2
3 action 3
```

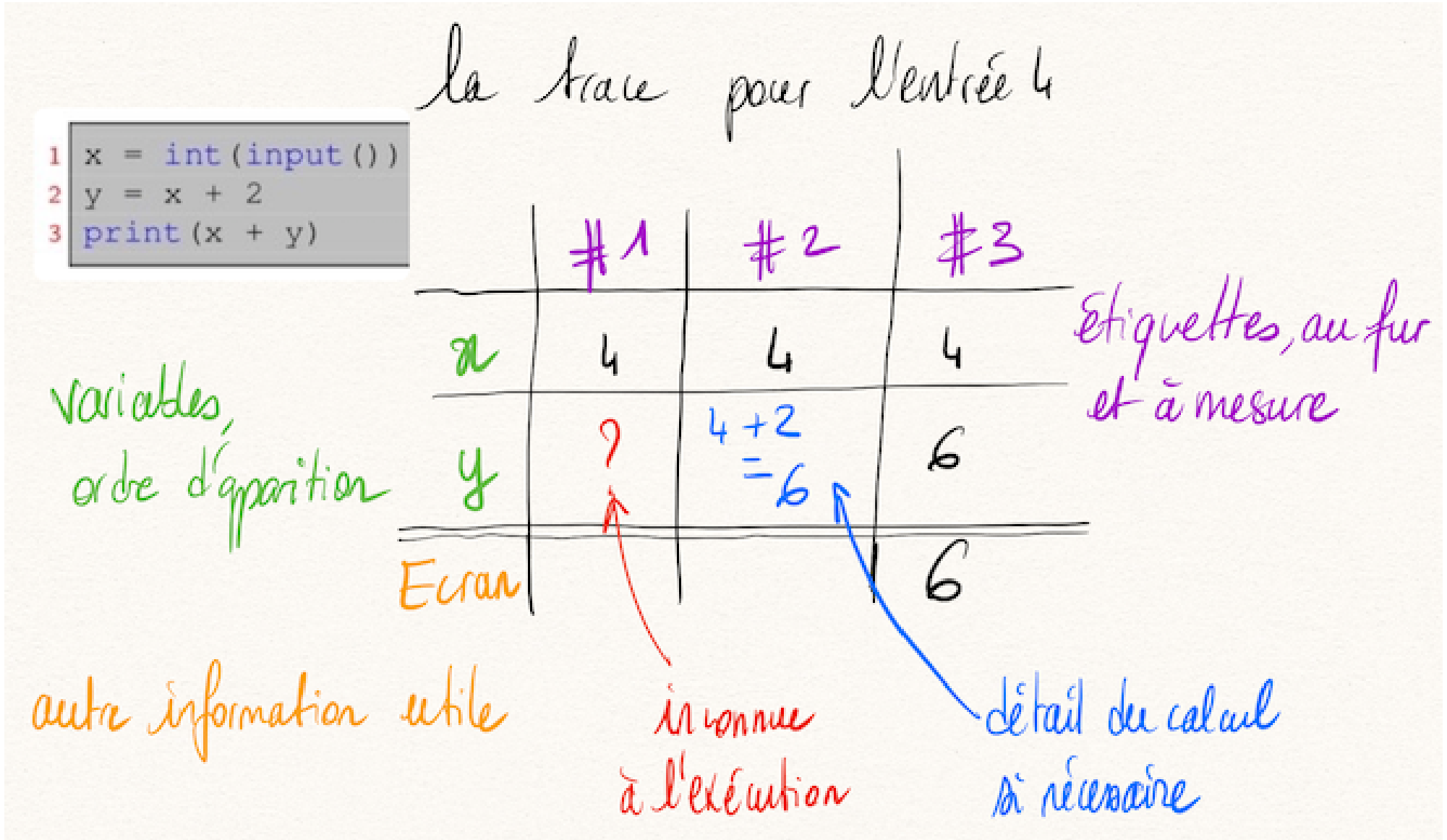
Se lit "exécuter action 1, puis action 2, puis action 3"

Séquence : exemple

```
1 x = int(input("x?"))
2 y = x + 2
3 print(x + y)
```

Pas à pas... représentation (trace ou tableau de situation) :

```
1 x = int(input())
2 y = x + 2
3 print(x + y)
```



Sélection

La sélection permet de choisir une action en fonction d'une condition

```
1 if <condition> :
2     <code >
```

Se lit "si condition s'évalue à vrai (True) alors exécuter code"

```
1 if <condition> :
2     <code1>
3 else :
4     <code2>
```

Se lit "si condition s'évalue à vrai (True) alors exécuter code1, sinon (condition qui s'évalue à False) exécuter code 2.
L'indentation sert de séparateur de bloc (`begin...end ou {}`)
`if` et `else` sont des mots clés réservés de python.

Sélection

Les opérateurs booléens sont :

```
== != < <= > >= or and not
```

Exemple de <condition> :

```
if x>0 :
if x>0 and y<0 :
if fonction(x,y,2+z)+7==28 :
...
```

On peut aussi ajouter `elif`, le "sinon si" :

```
if <condition1> :
  <code1>
elif <condition2> :
  <code2>
else : #facultatif
  <code3>
```

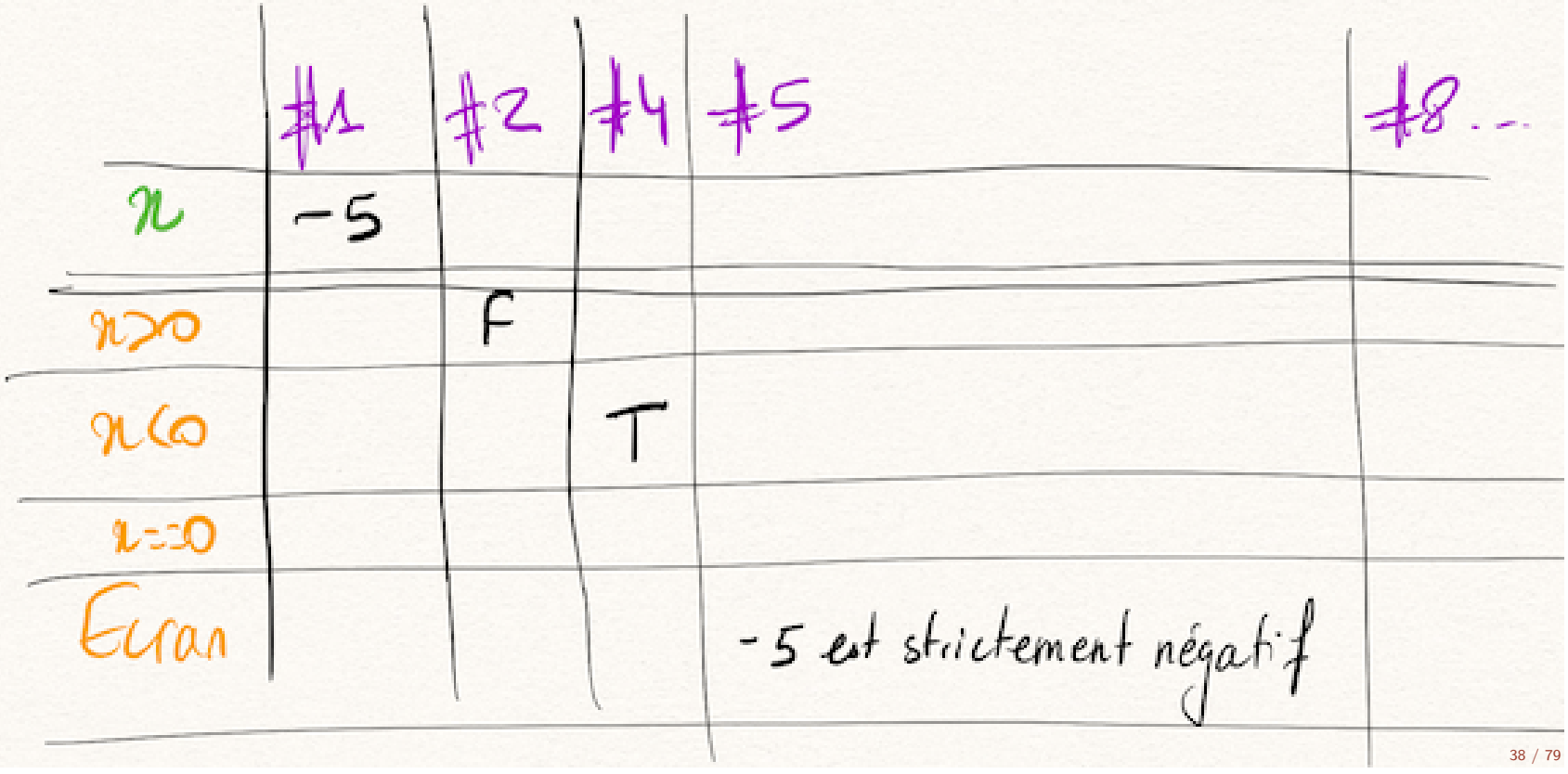
Sélection : exemple

```
1 x = int(input("x?"))
2 if x>0 :
3     print(x, "est strictement positif")
4 elif x<0 :
5     print(x, "est strictement négatif")
6 else :
7     print(x, "est nul")
```

Pas à pas... représentation :

```
1 x = int(input())
2 if x>0 :
3     print(x, "est strictement positif")
4 elif x<0 :
5     print(x, "est strictement négatif")
6 else :
7     print(x, "est nul")
```

la trace pour l'entier -5



Sélection

Exercice sur les booléens...

Répétition [2]

C'est une structure de contrôle qui permet de répéter un bloc d'instructions tant qu'une condition est vraie.

```
1 while <condition> :
2     <code >
```

Se lit "tant que condition s'évalue à vrai (True) exécuter code".

- Chaque exécution du bloc code est une *itération*.
- condition est une *expression logique* (booléenne). Cette condition doit avoir une valeur avant d'arriver au while.
- code représente le traitement à répéter et doit modifier la condition pour espérer terminer (erreur de conception).
- `while` est un mot clé réservé de python.

2. Débat possible sur l'introduction du `for` avant ou après le `while`...

Les 3 "essentiels" d'une boucle while

- Initialisation
- Condition
- Modification (des variables concernées par la condition)

A RABACHER ! ! !

Méthodologie : comment procéder ?

- Mauvaise méthode : se jeter sur le clavier, écrire un code erroné, le corriger, le corriger, le corriger,...
- Approche
  - Quelle est l'idée générale de l'algorithme ?
  - C'est ce qu'on appelle le *modèle de solution*
  - Pour le réaliser, se poser les questions :
    - Corps de la boucle : Quel est le *schéma répétitif* ?
    - Condition de la boucle : A quoi veut-on *aboutir* ?
    - Modification : *Comment passer "au suivant"* ?
    - Initialisation : *Comment démarrer* ?

Conseil :

- mettre la/les instructions de modification en fin de boucle.

Répétition : exemple

```
1 #somme des entiers tant que entier saisi non nul
2 #quantité d'itérations inconnue (introduire un booléen ?)
3 x = int(input("x?"))
4 somme = 0
5 while x>0 :
6     somme = somme + x
7     x = int(input("x?"))
8 print(somme)
```

```
1 #somme de n entiers saisis, n donné au début
2 #quantité d'itérations connue
3 n = int(input("n?"))
4 somme = 0
5 while n>0 :
6     x = int(input("x?"))
7     somme = somme + x
8     n = n - 1
9 print(somme)
```

Répétition

Exercices sur des whiles simple et double conditions

Répétition for

C'est une structure de contrôle qui permet de répéter un bloc d'instructions autant de fois que précisé.

```
1 for <variable> in range(<depart>,<fin_exclu>):
2       <code>
```

Se lit "pour toutes valeurs de variable dans l'intervalle [*depart*, *fin_exclu*[, exécuter *successivement* le code".

- variable va prendre initialement la valeur de l'indice de départ puis incrémenter de 1
- si on ne précise pas, l'indice de départ est 0
- la partie < code > de la boucle for va être exécutée pour chacune des valeurs

Il faut être sûr de vouloir exécuter*fin_exclu* − *depart* fois le code, sinon il est plus judicieux d'utiliser un `while` avec une condition qui permette d'arrêter la boucle avant la fin.

Les 3 "essentiels" d'une boucle for

Ils sont cachés, mais existent bel et bien !

- Initialisation : la variable est initialisé à la valeur depart
- Condition : tant que la variable est inférieure (stricte) à `fin_exclu`
- Incrémentation : de manière "masquée", la valeur de variable prendra (systématiquement) la valeur prévue de l'itération.

Bonnes habitudes

- INTERDIRE DE MODIFIER variable dans le code
- `break` à bannir (autant faire une while)
- nommer variable i souvent pratique, rajouter un ptit mot derrière ijour (pour un ième numéro de jour) ou inom (pour un ième nom)...

Un `for` peut toujours s'écrire avec un `while`, mais pas l'inverse !

Répétition : exemple

```
1 #exemple précédent
2 #somme de n entiers saisis, n donné au début
3 n = int(input("n?"))
4 somme = 0
5 while n>0 :
6     x = int(input("x?"))
7     somme = somme + x
8     n = n - 1
9 print(somme)
```

```
1 #somme de n entiers saisis, n donné au début
2 n = int(input("n?"))
3 somme = 0
4 for i in range(0,n) :
5     x = int(input("x?"))
6     somme = somme + x
7 print(somme)
```

Répétition

Exercices sur for, sur mélange for/while

Motivation

- Ré-utiliser
- Généraliser
- Améliorer

Mise en garde

La décomposition en fonction ne semble pas toujours pertinente dans les débuts du programmeur... (petits programmes pour lesquels cela ne s'y prête pas).

- IMPOSER l'écriture Sous-Programme + Programme Principal dès que les fonctions sont introduites.

Syntaxe DEFINITION

```
1 def <nom>(<param1,param2,... paramN>):
2             <instructions>
```

Se lit "on définit la fonction nom ayant les paramètres param1, param2... param N qui va exécuter les instructions".
Les valeurs des paramètres ne seront connues qu'à *l'appel* de la fonction.
On écrit donc quelque chose de "généraliste". On peut dire "pour toutes valeurs de param1, de param2..."...
On appelle *corps* les instructions de la fonction.

Description

```
1 def f(x):
2     ''' description de f'''
3     return x
4
5 help(f) #comme pour toute fonction ou type ou class de python
```

```
1 Help on function f in module __main__:
2
3 f(x)
4     description de f
```

- Intérêt de la description : avoir une aide automatique pour chaque fonction avec l'instruction `help(<nom>)`
- Description simple qui mentionne l'objectif de la fonction, les paramètres et leur rôle
- Retour d'expérience de la description : échec car "trop" exigent et peu intéressant sur petits programmes.

Syntaxe APPEL

```
1 ...
2 <nom>(<arg1,arg2,...,argN>)
3 ...
```

Se lit "on appelle la fonction nom avec les valeurs des arguments arg1, arg2, ... argN".
Les instructions du corps de la fonction seront exécutées en remplaçant les paramètres par les valeurs des arguments correspondants.

Le return

- Si la fonction doit produire une valeur, `return` va permettre de sortir de la fonction, et remplacer l'appel de fonction par la valeur retournée. C'est la majorité des cas, sinon c'est souvent de l'affichage ou de la mise à jour de type complexe.
- S'il n'y a pas de `return` la fonction se termine après la dernière instruction (et renvoie `None`).
- Il est possible d'avoir plusieurs return dans une fonction, le premier rencontré sort définitivement, mais ce qui est "après" ne sera jamais exécuté.

Paramètres/Arguments

- Paramètres/nom de fonction : choisir des noms courts et explicites (sans accent), il y a des règles de nommage.
- A l'appel, fournir un argument pour chaque paramètre. Un argument peut être une valeur, une variable, une expression qui est évaluée pour être ainsi affectée au paramètre correspondant de la fonction.
- le passage par paramètre peut être vu comme une affectation

Paramètres/Arguments

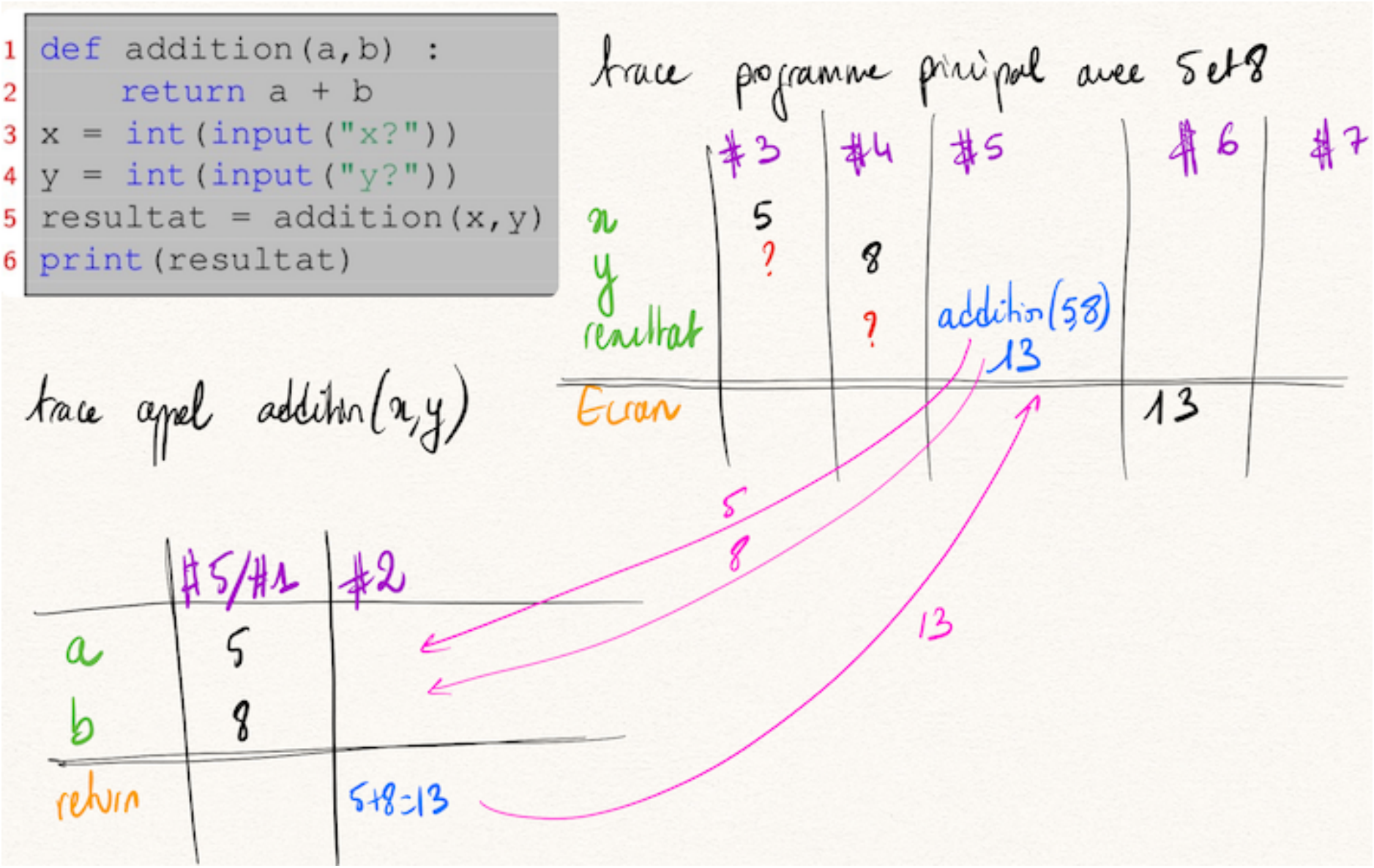
- les paramètres et les arguments sont indépendants : si la fonction modifie un paramètre, elle ne modifie pas l'argument correspondant
- les variables et paramètres de la fonction sont définis (et existent) uniquement pour l'exécution de la fonction, on les appelle des variables *locales* à la fonction. Elles n'ont pas d'existence en dehors de l'exécution de la fonction.
- Remarque : s'il n'y a pas de paramètres, l'appel se fait avec des parenthèses vides : `def fonction_sans_param(): ...`

Fonction : exemple

```
1 def addition(a,b) :
2     return a + b
3 x = int(input("x?"))
4 y = int(input("y?"))
5 resultat = addition(x,y)
6 print(resultat)
```

Pas à pas... représentation :

```
1 def addition(a,b) :
2     return a + b
3 x = int(input("x?"))
4 y = int(input("y?"))
5 resultat = addition(x,y)
6 print(resultat)
```



Fonction

On peut composer les appels de fonctions

```
1  #definition de la fonction
2  def minimum(a,b):
3      """Minimum de deux nombres a et b"""
4      # calcul
5      if (a<b):
6          min=a
7      else:
8          min=b
9      return min
10
11 #exemple d'appel
12 print(minimum(5,6))
13 print(minimum(4,minimum(1,6)))
```

Comportement variables, paramètres, arguments

Quelques exercices ?

```
1  def fonction_z1(a):
2      a=8
3      return 7
4
5  x = 20
6  y = fonction_z1(x)
7  print(x)
8  ####
9  def fonction_z2(a):
10     b=8
11     return 7
12
13 b = 20
14 y = fonction_z2(b)
15 print(b)
16 ####
17 a=10
18 def fonction_z3(a,b):
19     return a+b
20
21 a=3
22 a=fonction_z3(a,5)
23 print(a)
```

Fonctions

Exercices sur les fonctions

Premiers types complexes

Premiers types complexes

- Listes
- Mutabilité
- Tuples

Définition

- Une liste est une séquence d'éléments (l'ordre est important)
- La position d'un élément dans une liste est appelé son *indice*
- les indices commencent à 0
- On peut accéder à l'élément de tel ou tel indice
- On peut modifier une liste : c'est un objet *mutable* (contrairement à une chaîne de caractère, un entier...)
  - modification, ajout ou suppression d'éléments
- Attention, suivant les langages on peut ou pas modifier la taille, et/ou accéder facilement à tous les éléments

Bonnes habitudes

- ne pas faire `l=["a,"b","c"]` : le "l" est trop court, trop similaire au "I"
- dans les exemples, éviter de faire `liste = [1,2,3]` : confusion indice/élement
- Vocabulaire : éviter "ième élément"... parce que les indices commencent à 0.

Syntaxe d'affectation (liste complète)

```
1  liste1 = [4, 75, 12]
2  liste2 = [3.1, 0.5, -1.3]
3  liste3 = ["a", "b", "c"]
4
5  # on peut mélanger les types (déconseillé car pas possible dans
       tous les langages)
6  liste4 = ["a", 2, -3.5]
7  liste5 = ["a", 2, -3.5, [1, 2, "a"]]
8
9  #liste vide
10 liste6 = []
```

Syntaxe d'affectation/accès à un indice donné

```
1 liste1 = [4, 75, 12]
2 contenu_indice_i = liste1[1] #pour accéder à 75
```

Opérations

- la longueur d'une liste est obtenue par `len(liste)` : (NB : en anglais, "longueur" c'est "length")
- `len(liste1)` vaut 3
- le dernier élément d'une liste est `liste[len(liste)-1]`, raccourci syntaxique `liste[-1]`
- si on essaie d'accéder à `liste[len(liste)]` on obtient le message d'erreur `IndexError: list index out of range`

Parcours d'une liste

Pour parcourir tous les éléments d'une liste :

```
1 for <variable> in <liste>:
2       <code qui utilise la variable>
```

- La variable va prendre successivement comme valeur chacun des éléments de la liste
- l'ordre par défaut est de l'indice 0 au dernier
- la partie < code > de la boucle for va être exécutée pour chacune des valeurs

=> il faut être sûr de vouloir parcourir *toute* la liste, sinon utiliser un `while` avec une condition qui permette d'arrêter la boucle avant la fin.

Exemple

```
1 for elem in [1, 2, 4, 16]:
2     print("Element :", elem)
```

```
1 Element : 1
2 Element : 2
3 Element : 4
4 Element : 16
```

Mise en garde

- Pour éviter les confusions entre les `for`, rigueur du nommage : `ijour, imois` pour des indices ; `elem, jour, mois` pour des valeurs de listes.
- attention, ces formes n'existent pas dans tous les langages

Syntaxe : modification de listes

```
1 liste[p] = v! #affecte la valeur v à l'élément d'indice p
2 liste.append(e) # rajoute l'élément e à la fin de la liste
3 liste.insert(p, e) # insère e à la position p
4 del liste[p] # elève l'élément qui est à la position p
```

NB : `liste.append(e)` équivaut à `liste.insert(e, len(liste))`
Ces opérations sont des actions (comme une affectation) pas des fonctions, ce qui implique :

- qu'elles n'ont pas de résultats (si ce n'est `None`)
- qu'elles modifient la liste argument

Mutable/Immutable

- Un objet mutable est un objet qui peut être modifié, dont on peut changer les propriétés une fois qu'il a été défini.
- Les objets de type `bool`, `int`, `str`, `bytes`, `tuple`, `range`, `frozenset` sont immutables. Tous les autres types, les `list`, `dict`, `set`... ou les instances de vos propres classes sont des objets mutables.
- Une erreur courante est de confondre modification et réassignation.

```
1 x = 3
2 x = 7 # on réassigne, on ne modifie pas
3
4 liste = ["a","b","c"]
5 liste.append("d") # on modifie la liste
```

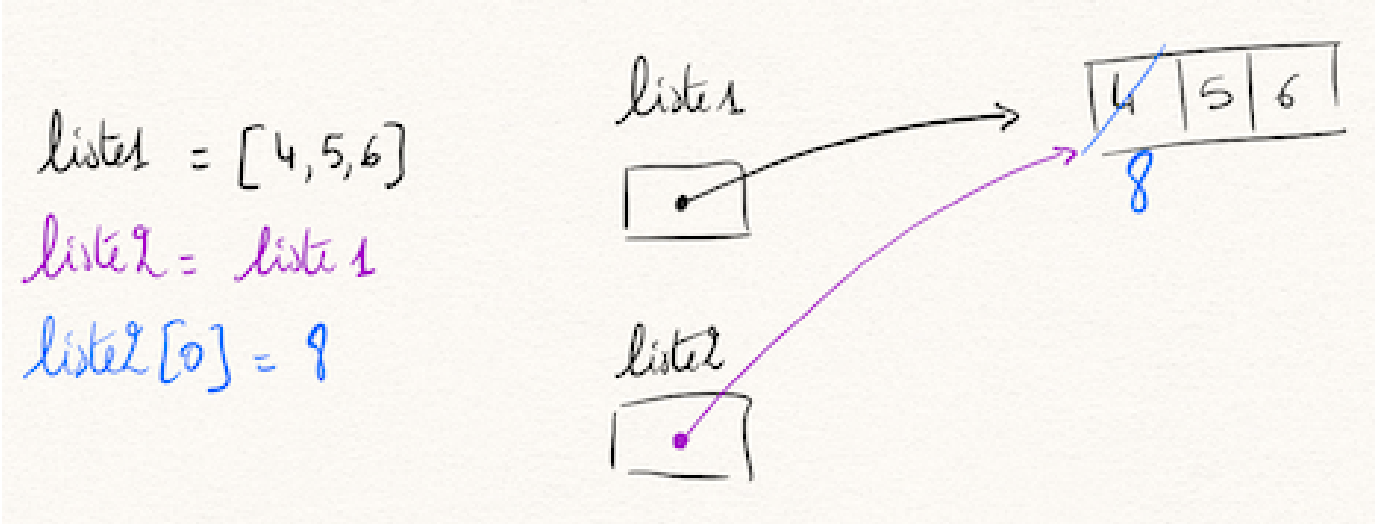
Conséquence - ATTENTION

- Une liste peut être "partagée" entre plusieurs variables.
  L'affectation doit être utilisée en connaissance de cause.

```
1 x = 3
2 y = x # on met le contenu de x dans y
3 y = 4
4 print(x,y) # 3 et 4
5
6 liste1 = ["a","b","c"]
7 liste2 = liste1
8 liste2.append("d") # on modifie la liste1 ET liste2
9 print(liste1,liste2) # ["a","b","c","d"] ["a","b","c","d"]
```

Premiers types complexes - Mutabilité

Représentation mémoire (schématique) :



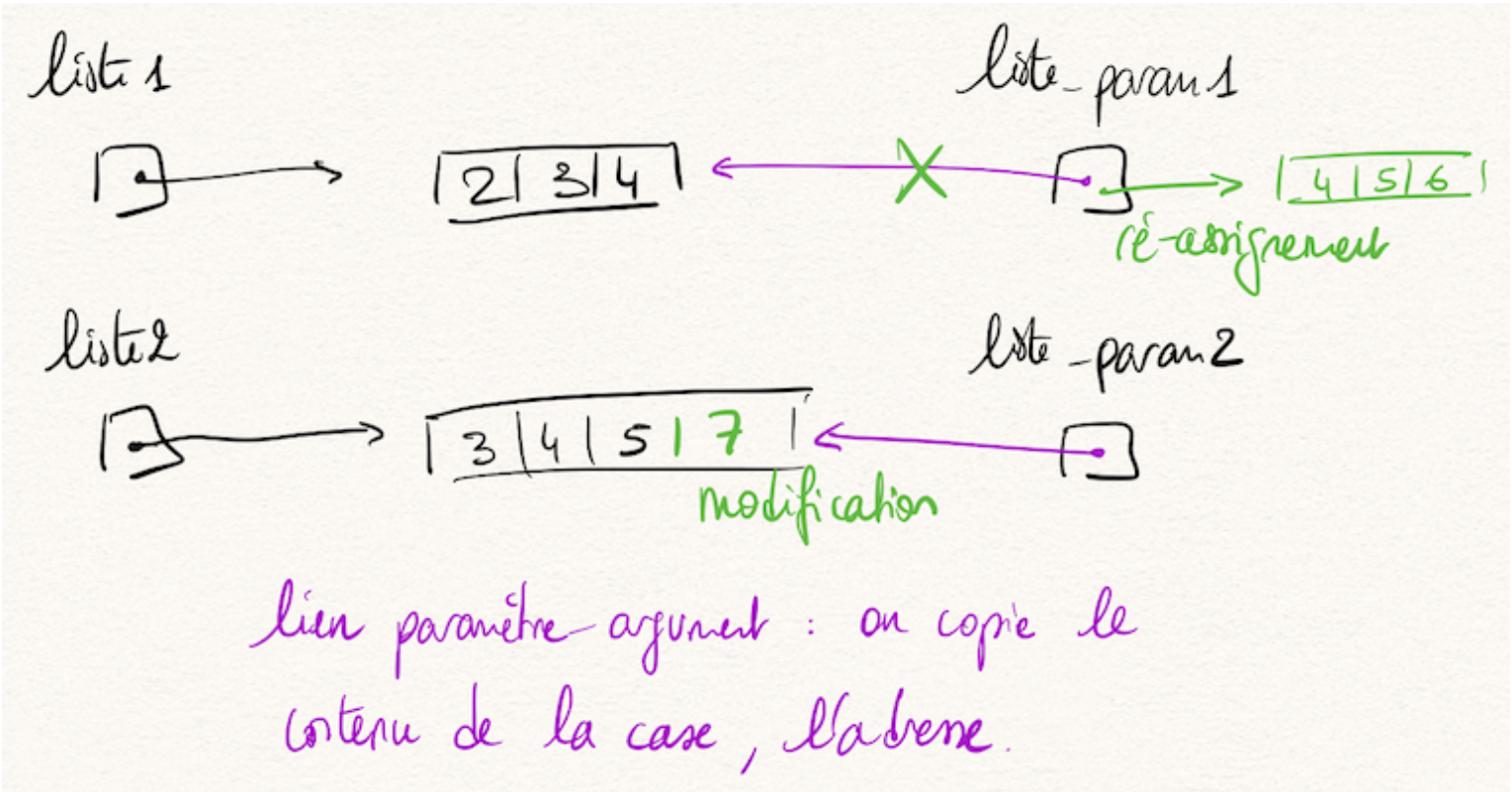
ou encore

Conséquence sur le passage de paramètre

- Différence de comportement entre paramètres mutables et immutables

```
1  def fonction_immutable(x):
2    x = 3
3    return 7
4  a = 5
5  b = fonction_immutable(a)
6  print(a,b) #5 7
7
8  def fonction_mutable(liste_param1, liste_param2):
9     liste_param1 = [4,5,6] # ré-assignement
10    liste_param2.append(7) # modification
11    return liste_param1[1]
12 liste1 = [2,3,4]
13 liste2 = [3,4,5]
14 a = fonction_mutable(liste1,liste2)
15 print(a,liste1,liste2) #5 [2, 3, 4] [3, 4, 5, 7]
```

Représentation mémoire (schématique) :



Autres opérations sur listes

- Addition ou concaténation

```
1 liste1 = [3,4]
2 liste3 = liste1 + liste1 #ré-assignement et opération
      d'addition
3 print(liste3) # [3,4,3,4]
```

- Tranche ou sous-liste

```
1 liste1 = [3,4,5,6,7,8]
2 sous_liste = liste1[2:4] #recopie les éléments d'indice 2
      à 4 exclu
3 print(sout_liste) # [5,6]
```

Attention à la notation mathématique [ ] qui induit en erreur : le dernier indice *exclu*.

Syntaxe

C'est un type complexe immutable.

```
1 tuple1 = ("abc",1,True)
2 tuple2 = "abc",1,True
3 tuple3 = tuple2 + (42,)
4
5 print(tuple1[0]) # abc
6 tuple1[0] = "d" #'tuple' object does not support item assignment
```

- Très naturel... sauf qu'on ne peut pas le modifier, il faut le ré-assigner
- Extrêmement utile pour les fonctions qui renvoient plusieurs valeurs, principe de pattern matching

Exemples

```
1  def fonction(a,b) :
2      return a+1,b+2
3
4  paire = fonction(3,4)
5  print("paire = ",paire)
6  val1, val2 = fonction(5,6)
7  print("val1 = ",val1)
8  print("val2 = ",val2)
9
10
11 liste = [(1,2) , (3,4) , (5,6)]
12 for prems,deuz in liste : #on peut isoler les élements du tuple
13         print("prems",prems)
```