R1.01 : Initiation au développement

Philippe POLET © 2024 : philippe.polet@uphf.fr

Objectifs du cours

- Participe à l'acquisition des compétences:
  - C1 : Développer des applications informatiques simples
  - C2 : Appréhender et construire des algorithmes
- Apprentissages critiques :
  - Implémenter des conceptions simples
  - Elaborer des conceptions simples
  - Faire des essais et évaluer leurs résultats en regard des spécifications
  - Analyser un problème avec méthode

Détail du programme national

- Algorithmes fondamentaux (structures simples, recherche d'un élément, parcours, tri…)
- Algorithmes sur les structures de données (itératifs et/ou récursifs)
- Manipulation de listes, tableaux, collections dynamiques, statiques (accès direct ou séquentiels), piles, files, structures
- Types abstraits de données simples : première approche de l'encapsulation
- Notions de modularité
- Premières notions de qualité (ex : nommage, assertions, documentation, sûreté de fonctionnement, jeu d'essais, performance…)
- Lecture/écriture de fichiers
- Présentation de la gestion de version

Règles de fonctionnement

- Présence « active » en cours, travaux dirigés et travaux pratiques
- Usage du téléphone interdit (règlement intérieur)
- PC portable interdit en cours, prise de notes uniquement sur papier
- Il faut suivre le cours, ce qui n'est pas compris en séance le sera difficilement après!
- En travaux dirigés, il ne faut pas être passif!
- Il faut avoir relu le cours avant d'aller en TD!
- Ne pas hésiter à multiplier les sources d'information (livres, internet, etc…)
- La programmation c'est comme le sport, il faut s'entrainer pour progresser (ce n'est pas en restant assis et en regardant les autres programmer qu'on progresse)

Introduction

CHAPITRE 1

Introduction à l'algorithmique

Un algorithme est une suite finie d'actions à réaliser pour résoudre un problème.

Nous avons des exemples d'algorithmes simples dans la vie de tous les jours:

- Recette de cuisine
- Notice de montage d'un meuble en kit.

Le niveau de détail et de précision des actions décrites dans un algorithme doit être de sorte que chaque action soit connue de l'exécutant et donc que l'interprétation de l'algorithme soit unique et ne prête pas à confusion.

L'algorithmique: notion de référentiel commun

Pour qu'un algorithme soit compréhensible il faut que celui qui rédige l'algorithme et l'exécutant partagent un ensemble de savoirs :

- Des objets (par exemple pour une notice de montage, si l'exécutant ne sait pas ce qu'est une vis, un marteau, une planche…)
- Des actions (percer, coller, visser,…)

C'est celui qui rédige l'algorithme qui doit se mettre à la portée de l'exécutant.

Il est évident que le rédacteur de l'algorithme doit savoir comment résoudre le problème.

L'algorithmique en informatique

Pour un ordinateur :

- Les objets sont : des données (essentiellement numériques puisque codées, au final, en binaire), ces données sont stockées, mémorisées dans des variables et utilisées dans des **expressions**.
- Les actions : correspondent à des opérations basiques : *+-/, des opérations pour stocker une valeur dans une variable… ces briques de bases sont ensuite assemblées pour construire des **fonctions** plus élaborées.

En informatique, l'algorithme est destiné au programmeur. Un algorithme décrit une nouvelle fonction qui permettra de réaliser un nouveau traitement. Généralement, cette fonction permet de calculer une valeur. Le programmeur codera l'algorithme dans un langage de programmation qui sera à son tour transcrit en langage machine.

L'important: c'est les valeurs

En informatique on manipule des données. Ces données sont de natures diverses. La notion de type est primordiale.

Il faut bien comprendre qu'en informatique les données sont codées en binaire. Ce codage peut varier selon le langage utilisé. Prenons, par exemple, une valeur sur 1 octet (8 bits), la taille de cette valeur est 1, mais la même suite de bits peut être interprétée différemment.

N°du bit	7	6	5	4	3	2	1	0
Valeur du bit	1	0	0	1	1	0	1	0

Entier non signé $\mathbf{2^7 + 2^4 + 2^3 + 2^1 = 128 + 16 + 8 + 2 = 154}$

Entier signé le bit 7 est le bit de signe => il faut alors faire le complément à 1 =>

$\mathbf{-(1 + 2^6 + 2^5 + 2^2 + 2^0) = -(1 + 64 + 32 + 4 + 1) = -102}$

Vous verrez dans d'autres modules comment peuvent être représentés les nombres réels en binaires…

Il faut donc retenir qu'une valeur est codée selon son type et occupe un nombre de bits correspondant à son type

Exemple le type **int** en *Python*

Une valeur entière peut être de type **int** (pour ***integer***).

Le type **int** code la valeur en binaire sur 4 octets (32 bits) 31 bits sont utilisés pour la valeur absolue, et le $32^{ième}$ bit sert à indiquer le signe (0: positif, 1: négatif).

La plus grande valeur **int** possible est donc $2^{31} - 1\ soit$ 2 147 483 647

Attention le codage des nombres en informatique peut poser des problèmes!

Les types de bases couramment utilisés

Type algorithmique	Type en Python	Exemples de valeurs (en Python)
entier	int	3256 -59
réel	float	-26.987 89.39877 1.0
booléen	bool	True False (attention Majuscule)
Chaîne de caractères	str	'toto et titi' « 1 »
RIEN		None

Remarque : la valeur 1 est différente de la valeur '1'

Notion d'expression

- Une expression est une combinaison syntaxique qui sera, lors de l'exécution, évaluée et substituée par une valeur.
- Une expression est donc typée.
- Exemples d'expressions:
  - 1+2*3 : est une expression qui est de **type entier** et qui sera substituée par la valeur **7** lors de l'exécution.
  - 1.0+2*3 : est une expression qui est de **type réel** et qui sera substituée par la valeur **7.0** lors de l'exécution.

- Dans une expression on pourra avoir une combinaison d'opérations entre des valeurs, des variables et des fonctions (qu'on aura préalablement programmées)…
- Ex: 24+ calculComplexe(2.5*var1, 5 , sin(60))
- Cette expression va être évaluée au moment de l'exécution : d'abord calculer sin(60), puis 2.5 * la valeur stockée dans var1, ensuite lancer la fonction avec les valeurs précédemment calculées et enfin additionner le résultat avec 24

Notion de variable

- Il est souvent utile de pouvoir stocker (mémoriser) une valeur pour pouvoir la réutiliser ultérieurement.
- Une variable permet de stocker une valeur. On lui affecte une valeur.
- L'opérateur d'affectation en algorithmique est : ←
- par exemple `a ← b*2+3` : la variable **a** reçoit le résultat de l'évaluation de l'expression *« valeur contenue dans **b** fois 2 plus 3 »*

  - **Dans une expression le nom de la variable est remplacé par la valeur qu'elle stocke.**

- Une variable est associée à une zone de la mémoire de l'ordinateur. C'est dans cette zone que seront stockés les bits décrivant la valeur.
- Une variable possède donc des propriétés :
  - Un **nom** → identifiant permettant de manipuler la variable
  - Un **type** → indique comment interpréter la valeur et de connaître sa taille
  - Une **valeur** → qui pourra être utilisée dans des expressions
  - Une **adresse** mémoire → là où est stockée la valeur dans la mémoire de l'ordinateur

Dans un programme, déclarer une variable permet de lui donner un nom, définir son type et demander au système de lui allouer un espace mémoire.

Illustration : représentation mémoire

- Soient 2 variables
  - v1 : caractère
    - Taille : 1 octet
    - Valeur : 'A' (code ASCII 65)
  - v2 : entier
    - Taille : 4 octets
    - Valeur : 65

B7	B6	B5	B4	B3	B2	B1	B0	adresse
0	1	0	0	0	0	0	1	0 (adresse de v1)
0	1	0	0	0	0	0	1	1 (adresse de v2)
0	0	0	0	0	0	0	0	2
0	0	0	0	0	0	0	0	3
0	0	0	0	0	0	0	0	4
0	0	0	0	0	0	0	0	5
0	0	0	0	0	0	0	0	6
0	0	0	0	0	0	0	0	7
0	0	0	0	0	0	0	0	8
0	0	0	0	0	0	0	0	9
0	0	0	0	0	0	0	0	10

Notion de fonction

Comme en mathématiques, une fonction calcule une valeur à partir de paramètre(s).

En informatique, une fonction sera donc la suite d'instructions permettant de retourner une valeur à partir des paramètres d'appel.

On distingue donc deux parties :

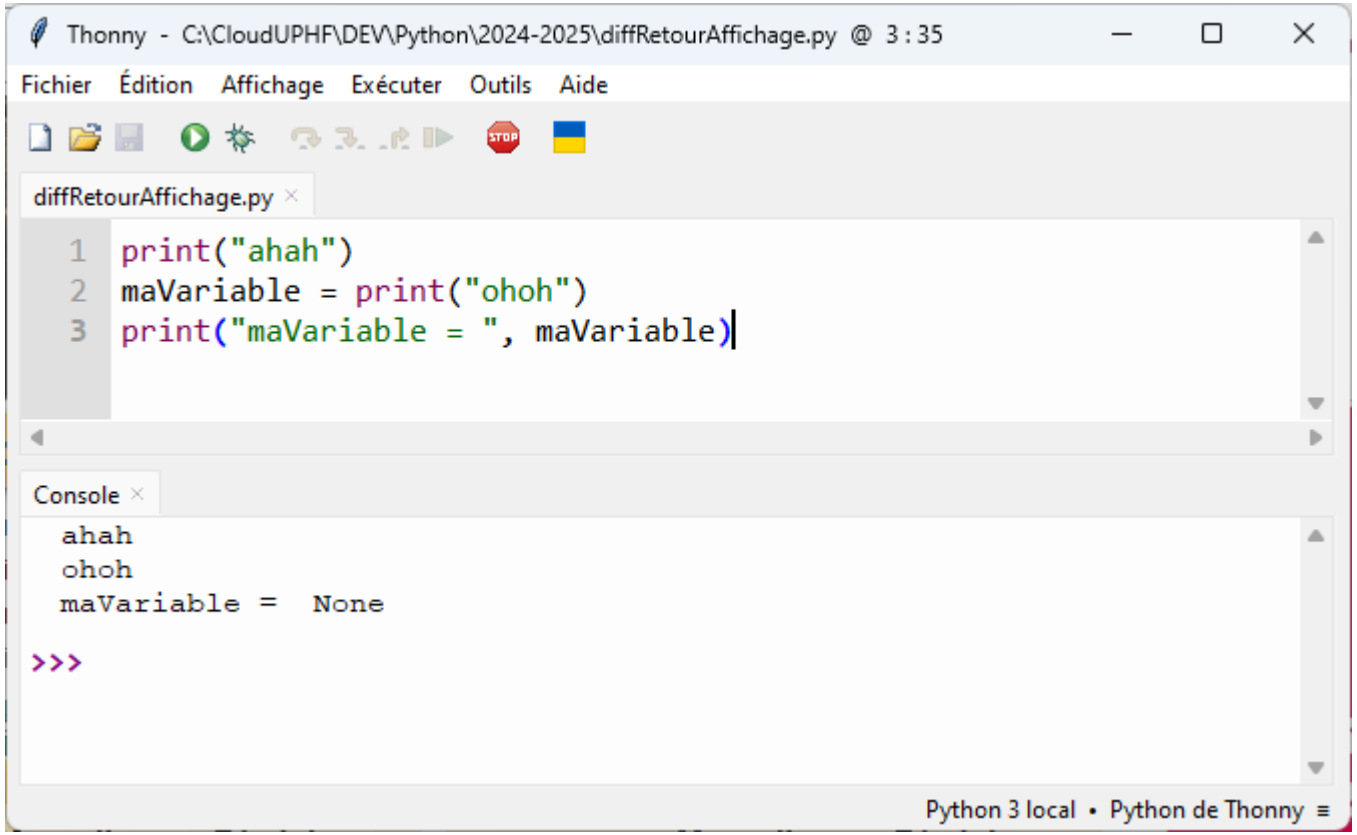
- L'**entête** de la fonction : renseigne le nom de la fonction, la liste et le type des paramètres ainsi que le type du résultat.
- Le **corps** de la fonction : définit les variables utilisées et décrit les instructions qui permettent de calculer le résultat.

Dans la plupart des langages il faut au moins une fonction: la fonction principale, qui est exécutée au démarrage du programme.

Quand une fonction est appelée dans une expression, elle est substituée par la valeur calculée.

Différence entre valeur retournée et valeur affichée

Retourner (ou renvoyer) une valeur c'est **différent** d'**afficher** une valeur !!!



Afficher : en python c'est la fonction **print** qui affiche une valeur

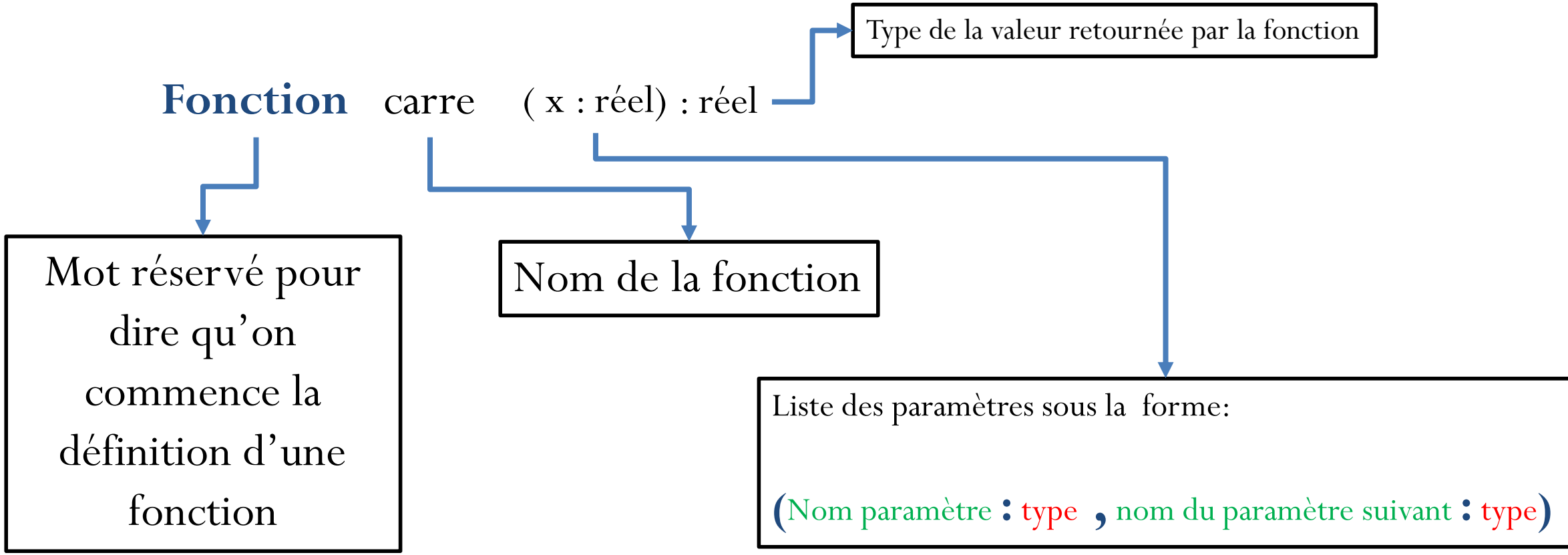
Typiquement la fonction **print** ne retourne rien (**None** en python)

Quand une fonction retourne une valeur, elle donne la valeur qui se substitue à son appel!

Notion d'algorithme

- Un algorithme doit servir de support à la réflexion mais également comme un outil de communication.
- Un algorithme est indépendant d'un langage. Le même algorithme pourra être traduit dans différents langages.
- Il est particulièrement adapté pour les langages impératifs (ADA, C, Pascal, python…).
- Un algorithme décrit les instructions d'une fonction.
- Commençons par une fonction simple: une fonction qui retourne la valeur (réelle) de son paramètre (réel) élevée au carré.

L'entête d'un algorithme (fonction)



Le corps d'une fonction en pseudo-code

On définit les variables locales (utilisées par la fonction) avec l'étiquette **variables locales :**

Chaque variable est alors listée avec son type de la forme : **NonVariable : Type**

Puis on liste les instructions à réaliser

La dernière ligne est : **Fin fonction**

Algorithme complet de la fonction ***carre***

Algorithme 1 la fonction carré

```
1: fonction CARRE(x : réel) : réel
2:     variables locales :                ▷ aucune variable locale
3:     renvoyer x * x
4: Fin fonction
```

Renvoyer : Mot réservé pour indiquer la valeur qui sera renvoyée par la fonction.
Par conséquent, dès que cette instruction est terminée l'algorithme est terminé!
Même s'il y a d'autres instructions après, elles ne seront pas exécutées.

Programme principal

- On veut un programme qui affiche 5 au carré, puis demande à l'utilisateur de saisir une valeur entière et affiche le carré de cette valeur. Il s'agit d'une procédure (le programme ne renvoie aucune donnée, et n'a pas de paramètre)

Algorithme 2 le programme principal

```
1: procédure PROGRAMME :
2:     variables locales :
3:     val : réel                    ▷ variable qui stocke la valeur saisie
4:     afficher(carre(5))
5:     afficher("saisir une valeur")
6:     val ← saisir()
7:     afficher(carre(val))
8: Fin procédure
```

Traduction d'un pseudo-code en Python

- Les variables ne sont pas déclarées en Python (***mais il est possible de le faire pour renseigner un analyseur de code***) c'est donc au programmeur de faire attention:
- L'opérateur d'affectation (←) est le = par exemple `v = 5;`
- Les fonctions sont déclarées de la manière suivante :

Optionnels : mais utiles pour la vérification des types

```
def carre(a:int)->int:
    return a*a
```

Traduction de la fonction ***carre***

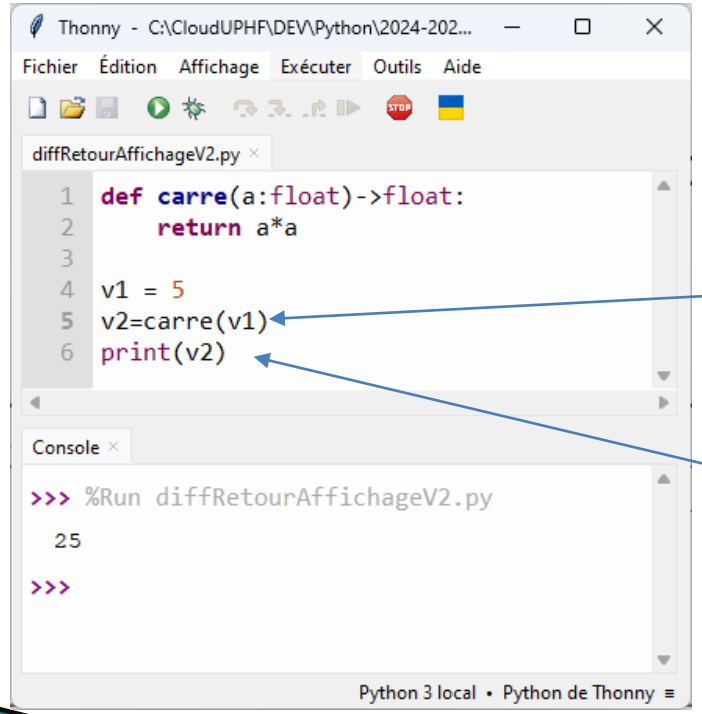
Pseudo-code

Algorithme 1 la fonction carré

1: **fonction** CARRE(x : *réel*) : réel
2: **variables locales :**
3: **renvoyer** $x * x$
4: **Fin fonction**

Python

```
def carre (a:float) ->float :
    return a * a
```



La fonction carre n'affiche rien!!!
Elle retourne une valeur qu'on stocke ici dans v2

On peut ensuite si on le souhaite afficher la valeur

Traduction du programme principal

Algorithme 2 le programme principal

```
1: procédure PROGRAMME :
2:     variables locales :
3:     val : réel                    ▷ variable qui stocke la valeur saisie
4:     afficher(carre(5))
5:     afficher("saisir une valeur")
6:     val ← saisir()
7:     afficher(carre(val))
8: Fin procédure
```

En Python, le programme commence par la première ligne du script qui n'est pas dans une fonction. Pour éviter de lancer le script d'un module importé on peut vérifier que les lignes appartiennent au script principal en faisant un test (nous verrons cela plus tard).

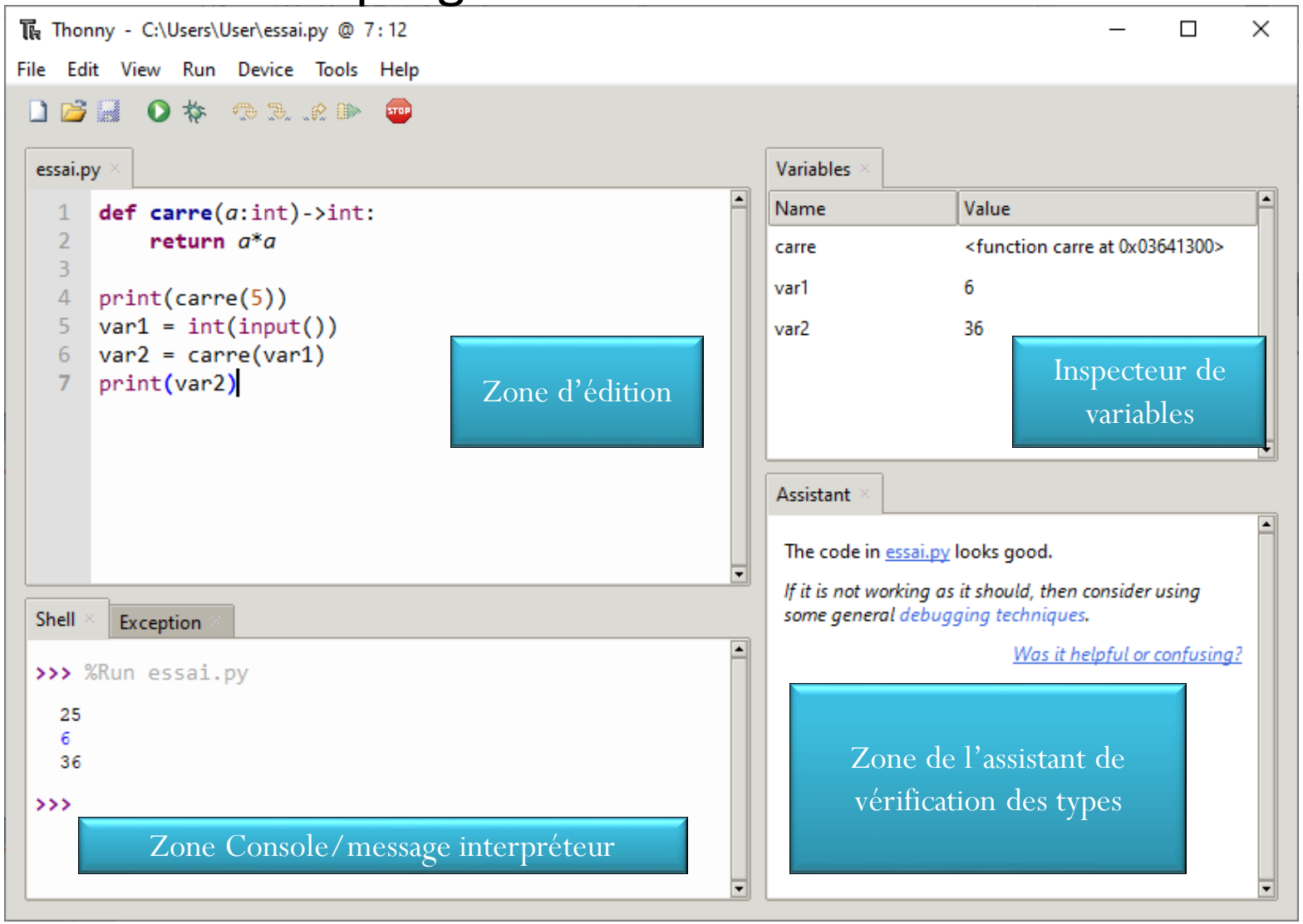
```
def carre(a:float)->float:
    return a*a

print(carre(5))
val = float(input())
print(carre(val))
```

- Pour afficher on utilise la fonction print()
- Pour récupérer une saisie on utilise la fonction input()
- La fonction input retourne une chaîne de caractères (str) que nous avons convertie, ici, en réel(float).

Environnement de programmation

Les fonctions square et le script principal sont décrits dans le même fichier (ici essai.py).
Un programme Python est écrit dans un fichier texte (éditable avec n'importe quel éditeur de texte) dont le nom se termine par **.py** !

Différents outils (EDI, environnement de développement intégré) existent. Nous utiliserons principalement **Thonny**.



Université Polytechnique HAUTS-DE-FRANCE

Branchement conditionnel

CHAPITRE 2

Un peu de logique : les booléens

- Une expression booléenne ne peut admettre que 2 valeurs VRAI ou FAUX (en Python True ou False)
- Opérations python simples renvoyant des booléens :

Opérateur	Expression	Signification
not	not expression	Non logique (not True = False / not False = True)
==	x == y	Égal
!=	x != y	Non égal
>	x > y	Plus grand que
<	x < y	Plus petit que
>=	x >= y	Plus grand ou égal à
<=	x <= y	Plus petit ou égal à
is	x is y	est le même objet
is not	x is not y	n'est pas le même objet

Operateurs sur les booléens :

- le **ET** (en Python **and**)
  - A ET B est Vrai si et seulement si A est Vrai et B est Vrai
- Le **OU** (en Python **or**)
  - A OU B est Vrai si au moins un des deux est Vrai

Traitement conditionnel

- Il est souvent nécessaire de réaliser des traitements particuliers (suite d'actions) selon certaines conditions.
- C'est ce que nous permet de faire la structure de contrôle « Si Alors Sinon »
  (en anglais : If Then Else, *ite*)

Le principe :

Si expression booléenne vraie

Alors on réalise le traitement A

Sinon on réalise le traitement B

Représentation en pseudo-code

Algorithme 4 exemple Si Alors Sinon

```
1: Si expression logique vraie alors
2:     faire action A
3:     faire action B
4: Sinon
5:     faire action C
6:     faire action D
7: Fin si
```

Actions réalisées si l'expression logique est **vraie**

Actions réalisées si l'expression logique est **fausse**

Code en Python

Algorithme 4 exemple Si Alors Sinon

```
1: Si expression logique vraie alors
2:     faire action A
3:     faire action B
4: Sinon
5:     faire action C
6:     faire action D
7: Fin si
```

```
if(expression):
        actionA
        ...
        actionB

else:
        actionC
        ...
        actionD
```

A noter :

1. l'expression booléenne est généralement placée entre **parenthèses (c'est une bonne habitude)**
2. Pas de mot clef pour le « alors »
3. Mot clef pour le bloc du « sinon » : **else**
4. **Les doubles points (:) sont OBLIGATOIRES**

Exemple d'application

- On souhaite écrire une fonction qui retourne l'état de l'eau en fonction de la température.
- La température sera donnée sous la forme d'un réel.
- L'état sera une valeur entière telle que:
  - ❖ 1 correspond à l'état solide,
  - ❖ 2 correspond à l'état liquide,
  - ❖ 3 correspond à l'état gazeux.

Quand la température est inférieure à 0°C, l'eau est à l'état solide. De 0 °C à 100°C, l'eau est à l'état liquide, et au-delà elle est à l'état gazeux.

Arbre Programmatique et code Python

Algorithme 5 Etat de l'eau

```
1:  fonction ETATEAU(t : réel) : entier
2:      Si t < 0 alors
3:          renvoyer 1
4:      Sinon
5:          Si t ≥ 0 ET t ≤ 100 alors
6:              renvoyer 2
7:          Sinon
8:              renvoyer 3
9:          Fin si
10:     Fin si
11: Fin fonction
```

```python
def etatEau(t:float)->int:
    if(t<0):
        return 1
    else:
        if((t>=0) and (t<=100)):
            return 2
        else:
            return 3
```

On remarquera l'imbrication des structures de contrôles. En pseudo, cette imbrication apparaît par l'indentation. En programmation, c'est le codeur qui doit faire un **effort** pour la rendre **visible**. Il est important d'adopter dès le début des règles d'écriture de code et notamment s'appliquer à **indenter** son code (d'ailleurs imposé par Python).

Intérêts de la décomposition modulaire

- Techniquement il est possible de tout écrire un programme dans la fonction principale ou script principal.

- Mais c'est une erreur, décomposer en fonctions permet:
  - ❖ d'améliorer la visibilité, donc la compréhension du code/algorithme,
  - ❖ de faciliter l'analyse: étymologiquement* analyser c'est décomposer!
  - ❖ de permettre la réutilisation,
  - ❖ de factoriser du code,
  - ❖ de permettre une maintenance/réactualisation du code simplifiée.

*Analyser : Nom formé à partir d'un terme grec « analusis » qui signifie « décomposition » , lui-même formé à partir d'un verbe simple "luein" qui signifie "décomposer".

Extension du problème

- On nous demande de réaliser un programme qui demande à l'utilisateur la température actuelle et qui affiche en fonction de cette température l'état physique de l'eau et de l'éthanol.
- Le but va être de factoriser du code et de réutiliser ce qu'on a déjà écrit…

Algorithme 5 Etat de l'eau

```
1:  fonction EtatEau(t : réel) : entier
2:      Si t < 0 alors
3:          renvoyer 1
4:      Sinon
5:          Si t ≥ 0 ET t ≤ 100 alors
6:              renvoyer 2
7:          Sinon
8:              renvoyer 3
9:          Fin si
10:     Fin si
11: Fin fonction
```

À garder!?

La fonction etatEthanol

Algorithme 6 Etat de l'éthanol

```
1:  fonction EtatEthanol(t : réel) : entier
2:      Si t < −117 alors
3:          renvoyer 1
4:      Sinon
5:          Si t ≥ −117 ET t ≤ 80 alors
6:              renvoyer 2
7:          Sinon
8:              renvoyer 3
9:          Fin si
10:     Fin si
11: Fin fonction
```

```python
def etatEthanol(t:float)->int:
    if(t<-117):
        return 1
    else:
        if((t>=-117) and (t<=80)):
            return 2
        else:
            return 3
```

On s'aperçoit que le code est très similaire au précédent seules les bornes de changement d'état on été modifiées…

Factorisation du code

- Imaginons qu'à l'avenir on nous demande également l'état du fer et du verre…
- Réalisons une fonction **générique**:

Algorithme 7 Etat Element

```
1:  fonction EtatElement(t : réel, bsol : réel, bgaz : réel) : entier
2:      Si t < −bsol alors
3:          renvoyer 1
4:      Sinon
5:          Si t ≥ bsol ET t ≤ bgaz alors
6:              renvoyer 2
7:          Sinon
8:              renvoyer 3
9:          Fin si
10:     Fin si
11: Fin fonction
```

```python
def etatElement(t:float,bsol:float,bgaz:float)->int:
    if(t<bsol):
        return 1
    else:
        if((t>=bsol) and (t<=bgaz)):
            return 2
        else:
            return 3
```

Modifications des fonctions précédentes

```
def etatEau(t:float)->int:
    if(t<0):
        return 1
    else:
        if((t>=0) and (t<=100)):
            return 2
        else:
            return 3
```

```
def etatEthanol(t:float)->int:
    if(t<-117):
        return 1
    else:
        if((t>=-117) and (t<=80)):
            return 2
        else:
            return 3
```

```
def etatElement(t:float,bsol:float,bgaz:float)->int:
    if(t<bsol):
        return 1
    else:
        if((t>=bsol) and (t<=bgaz)):
            return 2
        else:
            return 3

def etatEau(t:float)->int:
    return etatElement(t,0,100)

def etatEthanol(t:float)->int:
    return etatElement(t,-117,80)}
```

Afficher un état

Algorithme 8 affichage état

```
1:  fonction PRINTETAT(etat : entier) :
2:      Si etat = 1 alors
3:          afficher("solide")
4:      Sinon
5:          Si etat = 2 alors
6:              afficher("liquide")
7:          Sinon
8:              afficher("gazeux")
9:          Fin si
10:     Fin si
11: Fin fonction
```

```python
def printEtat(etat:int)->None:
    if(etat == 1):
        print('solide')
    else:
        if(etat == 2):
            print('liquide')
        else:
            print('gazeux')
```

Le programme principal

Dans un fichier python on peut simplement mettre des fonctions qui constitueront éventuellement une bibliothèque utile qui pourra être importée.
Le programme principal est donc un bout de script qui ne doit être lancé que si le script est lancé et pas simplement importé comme une bibliothèque.

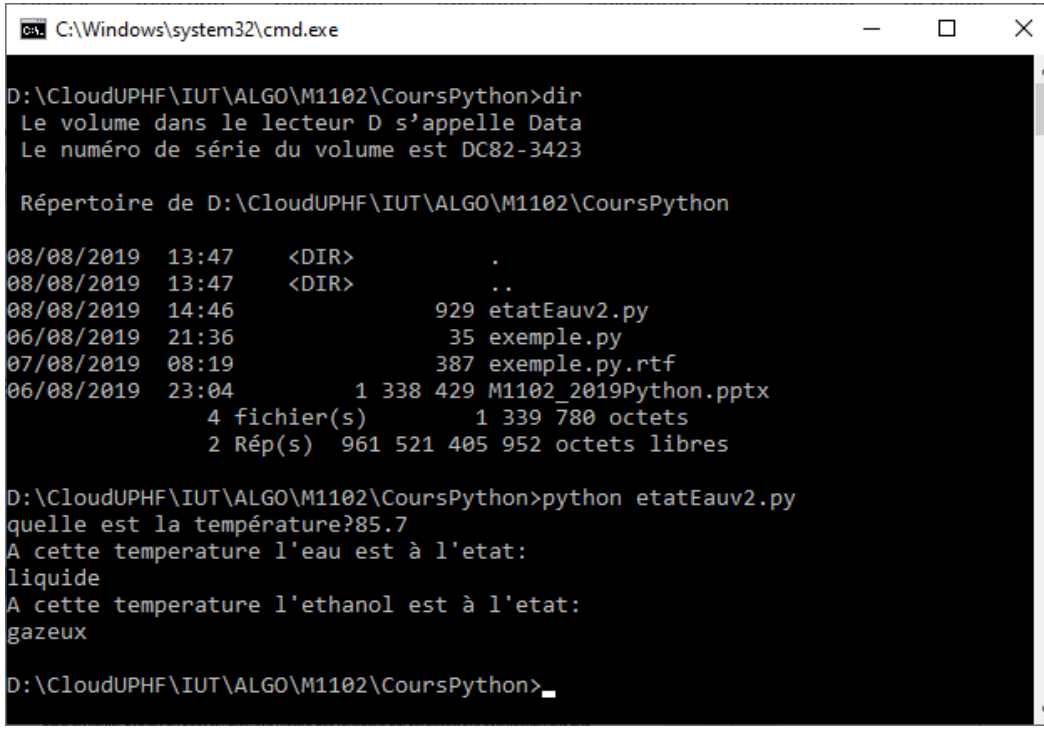
Python dispose d'une variable globale `__name__` qui contient soit le nom du module quand le fichier est chargé comme un module, soit la chaîne de caracteères `"__main__"` si le programme est utilisé de manière autonome. Par conséquent, il est possible de tester cette variable pour sélectionner le code qui ne doit être effectué que quand le fichier est exécuté en tant que programme.

On utilise pour cela la ligne de code suivante :

```
if __name__ == "__main__":
```

illustration

```
if __name__ == "__main__":
    temperature:float
    eau:int
    ethanol:int
    temperature = float(input('quelle est la température?'))
    eau = etatEau(temperature)
    print("A cette temperature l'eau est à l'etat:")
    printEtat(eau)
    ethanol = etatEthanol(temperature)
    print("A cette temperature l'ethanol est à l'etat:")
    printEtat(ethanol)
```



L'instruction `elif` en Python

```
def printRomain(n:int)->None:
    if(n==1):
        print('I')
    else:
        if(n==2):
            print('II')
        else:
            if(n==3):
                print('III')
            else:
                if(n==4):
                    print('IV')
                else:
                    if(n==5):
                        print('V')
                    else: print('??')
```

```
def printRomain2(n:int)->None:
    if(n==1):
        print('I')
    elif(n==2):
        print('II')
    elif(n==3):
        print('III')
    elif(n==4):
        print('IV')
    elif(n==5):
        print('V')
    else: print('??')
```

Université Polytechnique HAUTS-DE-FRANCE

Les boucles

CHAPITRE 3

Exemple de problème

- On désire écrire un algorithme d'une fonction qui admet en paramètre une valeur réelle (x) et une valeur entière positive (n) et qui retourne x^n.

Analyse :

Cas particulier $n = 0 \Rightarrow x^n = 1$

$$x^1 = x$$
$$x^2 = x * x$$
$$x^3 = x * x * x$$
$$\dots$$
$$x^n = x * \cdots * x$$

Pour calculer x^n on peut multiplier $x\ par\ lui\ même, n-1\ fois$

Répéter un nombre de fois donné un traitement

- Principe :
- On utilise une variable qui jouera le rôle de compteur.
  - On initialise le compteur à une valeur de départ
  - On réalise le traitement à répéter
  - À la fin le compteur est incrémenté (+1) et on compare la valeur du compteur avec la valeur de fin
    - Si on a atteint la valeur de fin on sort de la boucle et on poursuit le reste des instructions
    - Sinon on refait un cycle

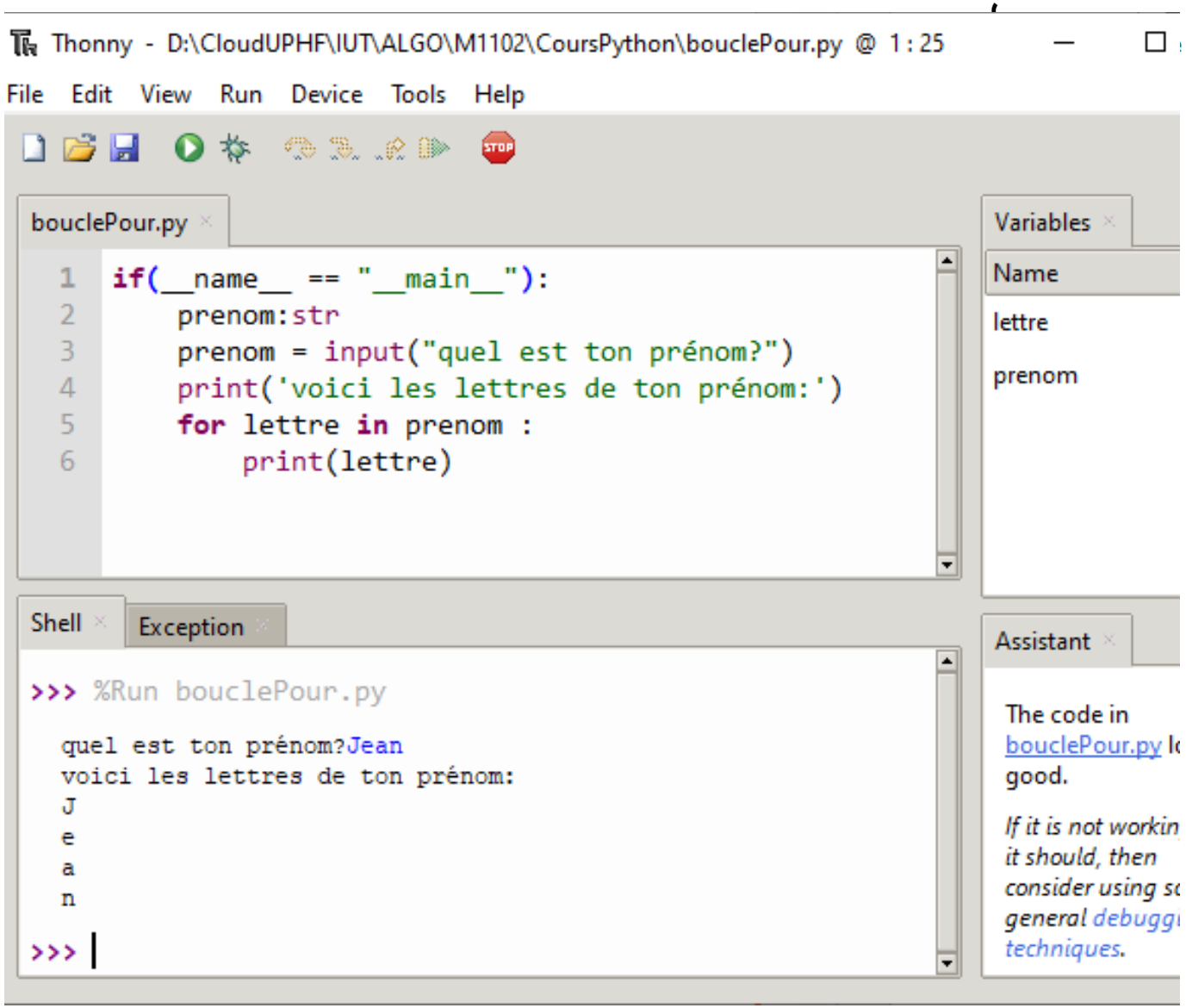
La boucle « pour » algorithmique

Algorithme 9 boucle pour

```
1: Pour i :1 à 5, faire :
2:     faire action A
3:     faire action B
4:     faire action C
5: Fin pour
```

La boucle **pour** en Python

- En Python, il n'existe pas vraiment de boucle « pour » au sens algorithmique. On utilise une boucle qui réalise un traitement pour toutes les valeurs contenues dans un ensemble (liste, chaînes de caractères…), nous reviendrons plus tard sur cette notion d'ensemble…
- Par exemple, une chaîne de caractères constitue un ensemble de caractères.
- L'instruction **for … in … :** permet de réaliser ce genre de boucle

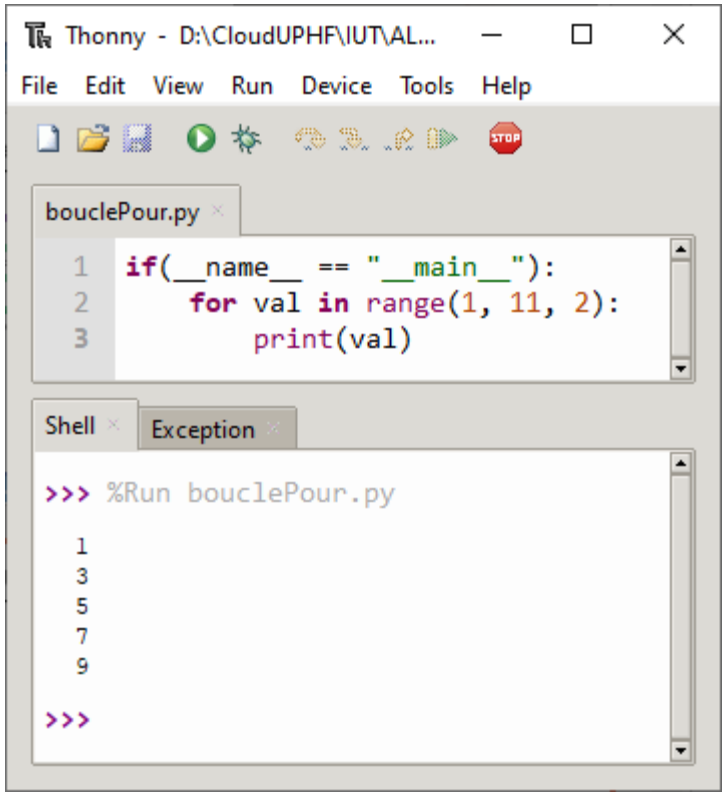


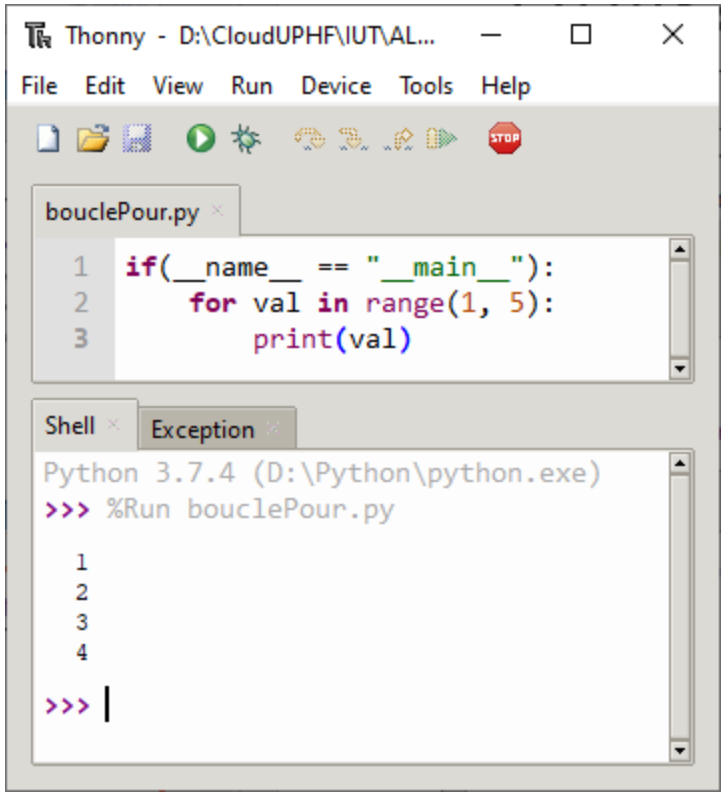
Le « type » (classe) range en Python

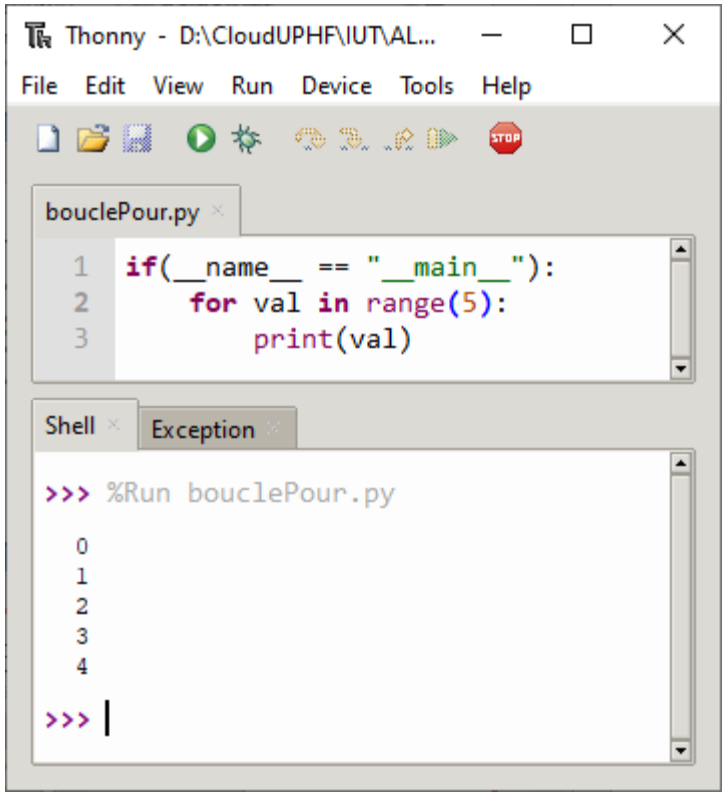
Un **range** est un ensemble d'entiers qui se suivent. On dispose de 3 méthodes pour créer un **range** :

- **range**(**debut**, **fin**, **pas**) : décrit une suite de valeurs commençant par la valeur **debut**, les suivantes sont augmentées de **pas** jusqu'à **fin** non inclus.
- **range**(**debut**, **fin**) : décrit une suite de valeurs commençant par la valeur **debut**, les suivantes sont augmentées de **1** jusqu'à **fin** non inclus.
- **range**(**fin**) : décrit une suite de valeurs commençant par la valeur **0**, les suivantes sont augmentées de **1** jusqu'à **fin** non inclus.

Utilisation d'un range dans une boucle for



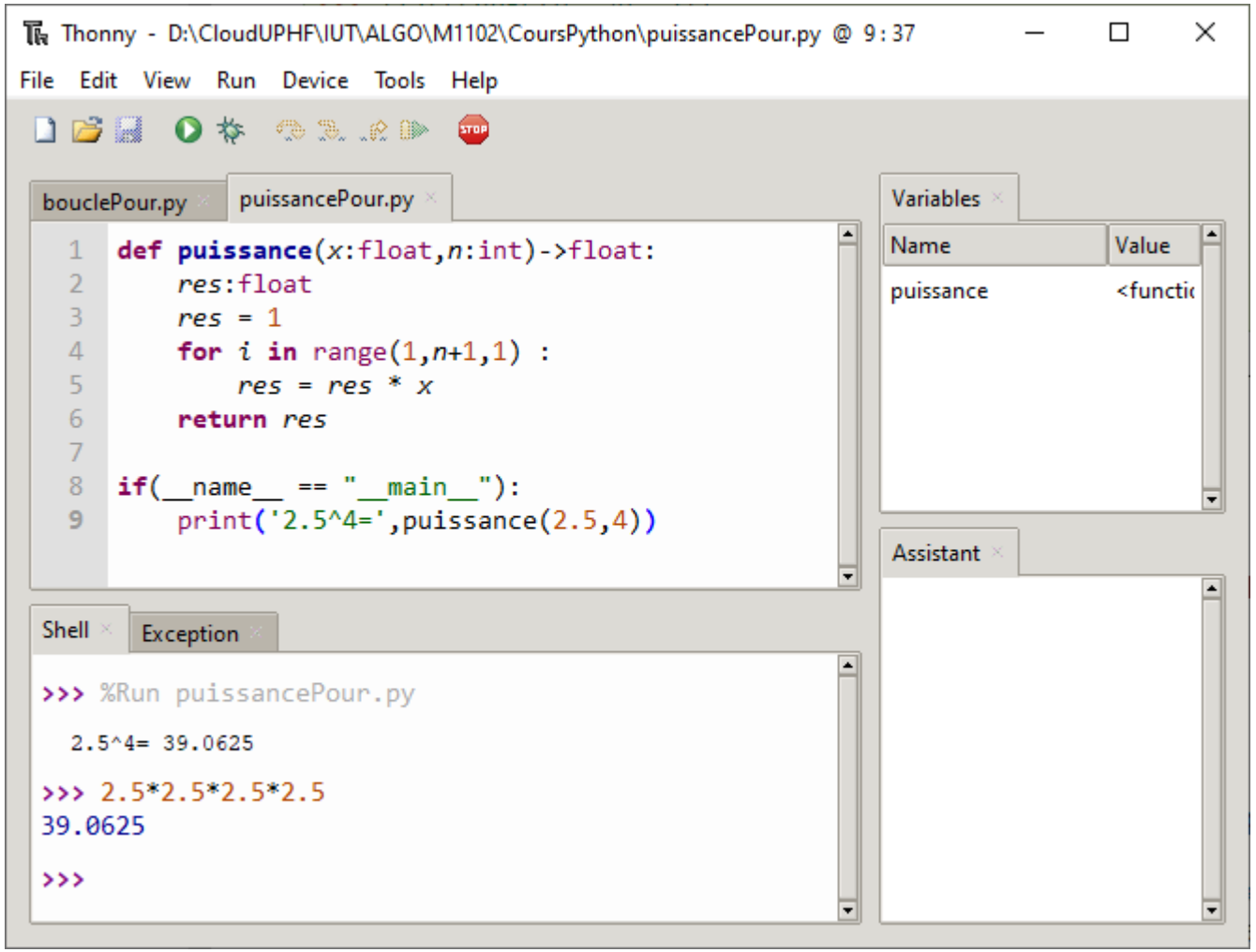




La fonction puissance avec une boucle ***pour***

Algorithme 3 la fonction puissance Pour

```
1:  fonction PUISSANCE(x : réel, n : entier) : réel
2:      variables locales :
3:      i : entier
4:      res : réel
5:      res ← 1
6:      Pour  i : 1 à n , faire :
7:          res ← res * x
8:      Fin pour
9:      renvoyer res
10: Fin fonction
```



Répétition conditionnelle

- Principe : répéter un traitement tant qu'une expression est vraie
- En début de cycle on évalue une expression booléenne:
- Si elle vaut VRAI
  - On réalise le traitement de la boucle
- Sinon on sort de la boucle

La boucle « tant que »

Algorithme 11 boucle Tant Que

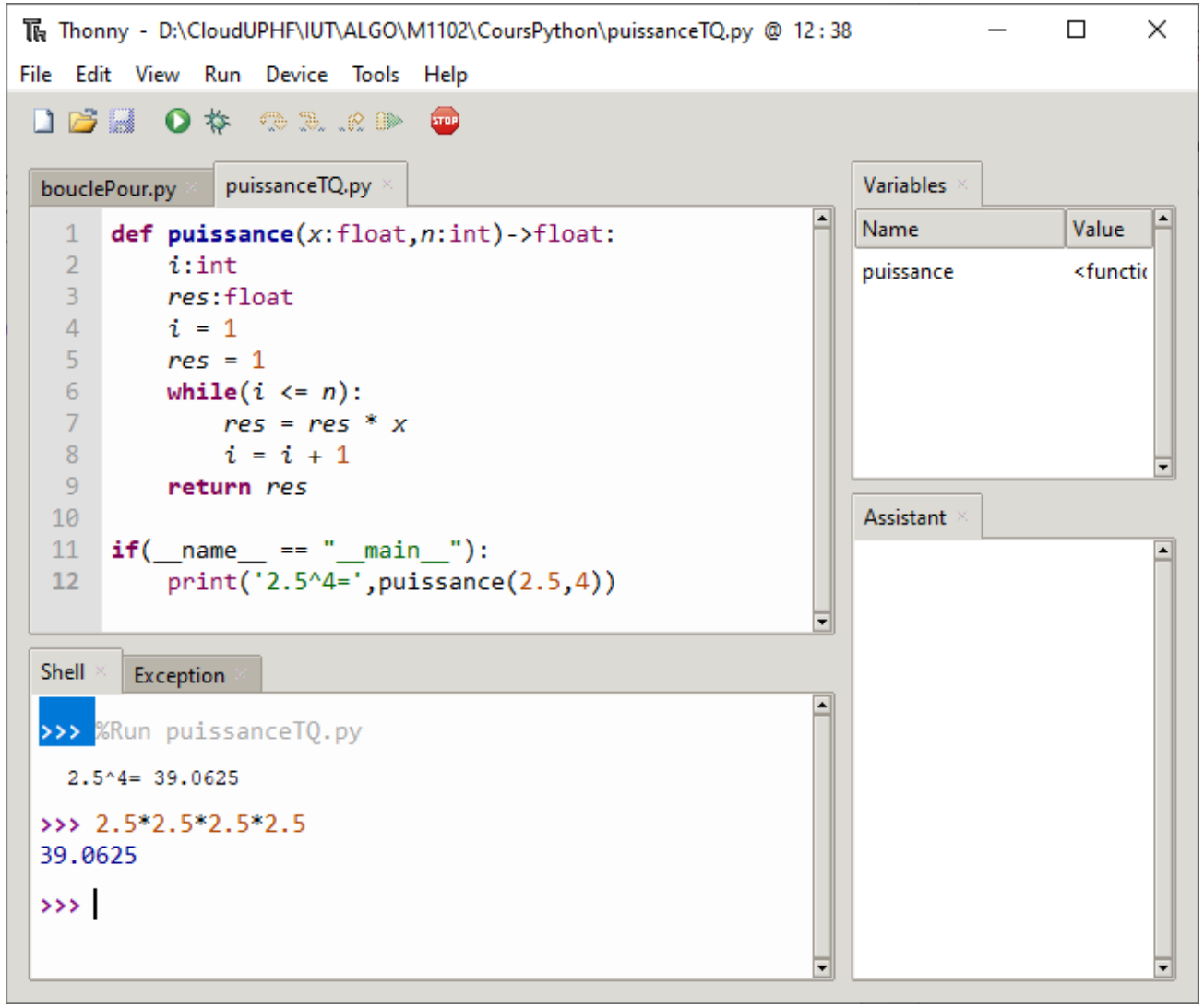
```
1: Tant que expr, faire :
2:     faire action A
3:     faire action B
4:     faire action C
5: Fin tant que
```

```
while(expr):
        action1
        action2
        action3
```

La fonction puissance avec une boucle ***tant que***

Algorithme 4 la fonction puissance Tant Que

```
1:  fonction PUISSANCE(x : réel, n : entier) : réel
2:      variables locales :
3:      i : entier
4:      res : réel
5:      res ← 1
6:      i ← 1
7:      Tant que  i ≤ n , faire :
8:          res ← res * x
9:          i ← i+1     Fin tant que
10:         renvoyer res
12:     Fin fonction
```



Bien écrire une boucle ***Tant Que***

Il est important de respecter des règles quand on utilise des boucles ***Tant Que***:

- Il faut s'assurer que l'algorithme permet d'initialiser l'expression de la condition:
  - Soit avec la valeur des paramètres reçus
  - Soit par une (série d') action(s) d'initialisation
- D'avoir, au moins, une action permettant de faire évoluer la valeur de l'expression

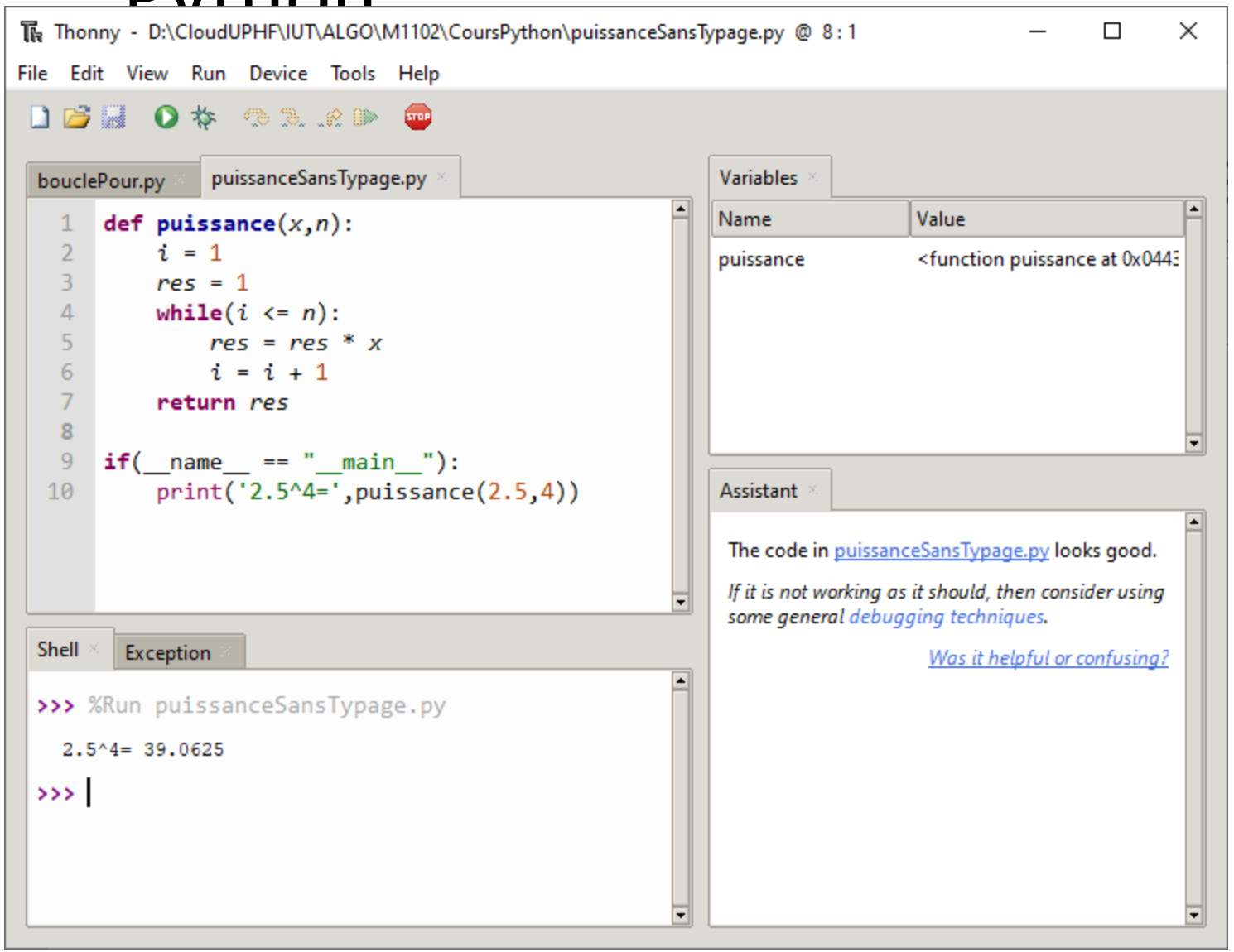
Dans le cas contraire, on risque de ne jamais « entrer » dans la boucle ou de ne jamais en sortir!!!

Algorithme 12 boucle Tant Que bien écrite

```
1: initialiser l'expr
2: Tant que expr, faire :
3:     actions à répéter
4:     action susceptible de faire évoluer expr
5: Fin tant que
```

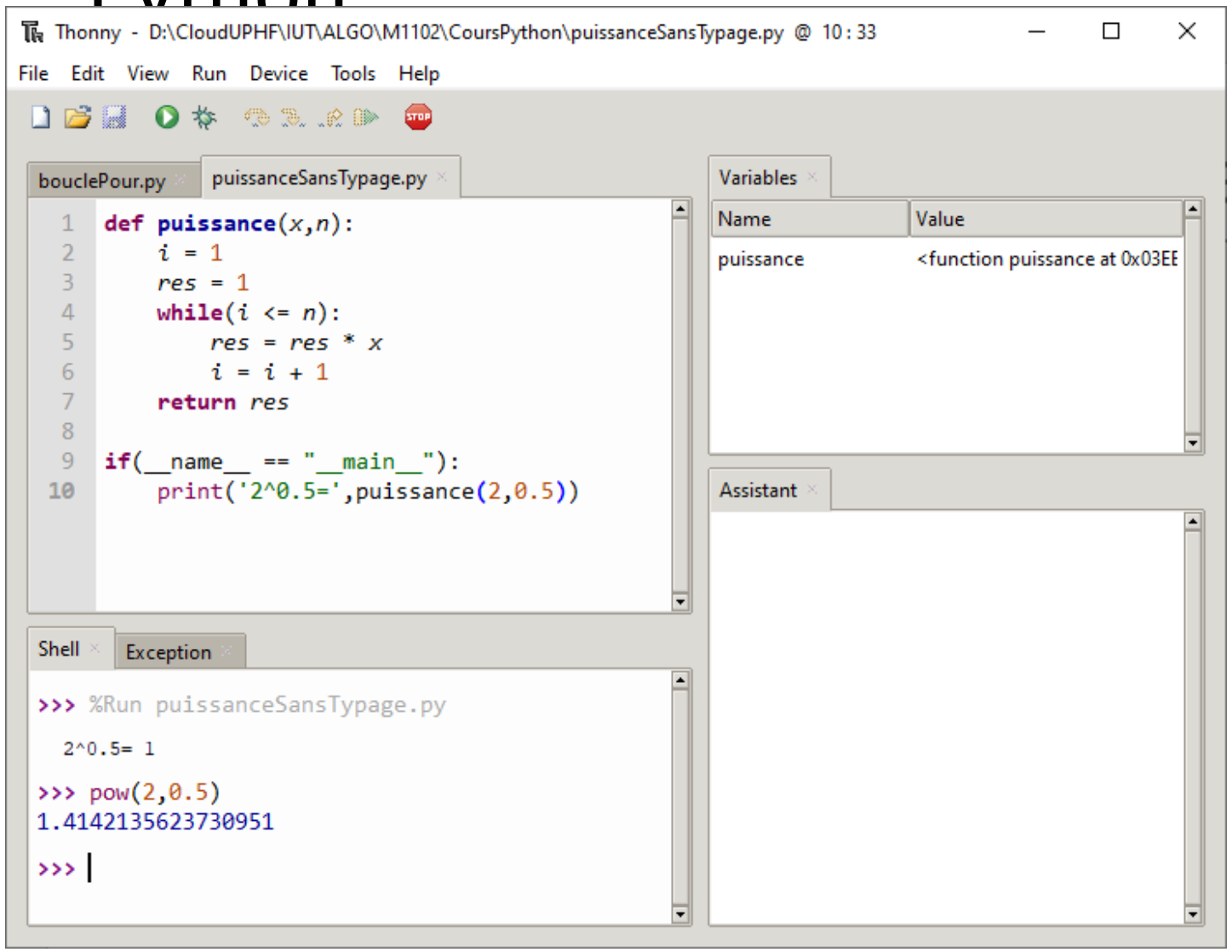
Remarque sur l'importance d'annoter les types en Python

- Le code suivant est correct en Python :
- C'est d'ailleurs sans annotation de type que vous trouverez la plupart des codes Python dans les différents ouvrages et sur les sites internet.
- Pourtant, cela est dangereux et peut être source de bugs.
- L 'annotation va permettre des vérifications qui seront utiles lors des tests.

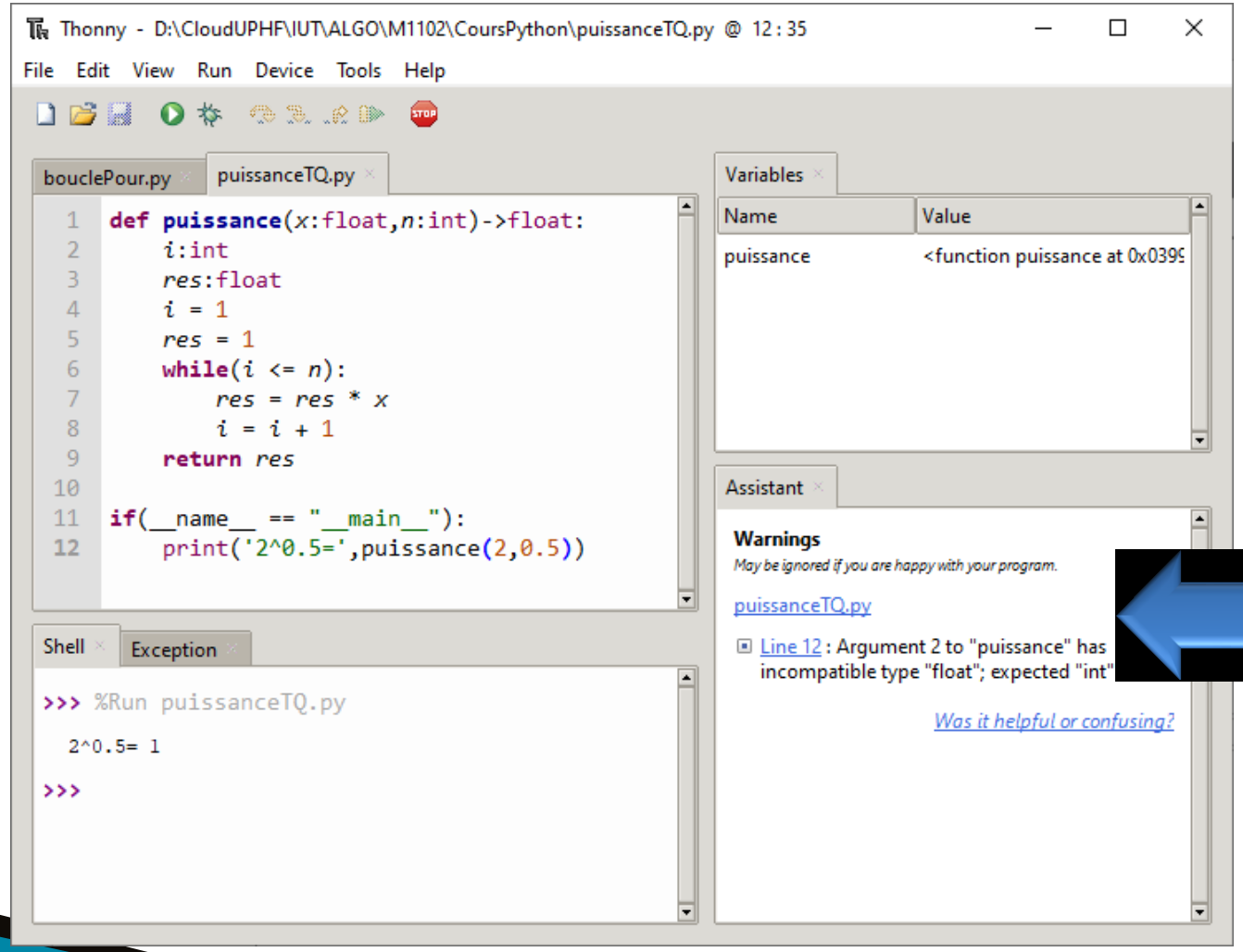


Remarque sur l'importance d'annoter les types en Python

- Le code suivant est correct en Python :
- Retour erroné si l'appel n'est pas correct:
- 2^0,5 n'est pas égal à 1: expliquez le résultat
- Il existe en Python la fonction « puissance », c'est la fonction **pow** et elle admet des puissances réelles.
- En fait 2^0,5 correspond à $\sqrt{2}$



Intérêt des annotations de type



L'assistant (mypy check) est en mesure d'identifier le problème

Université Polytechnique HAUTS-DE-FRANCE

Les données composites(introduction)

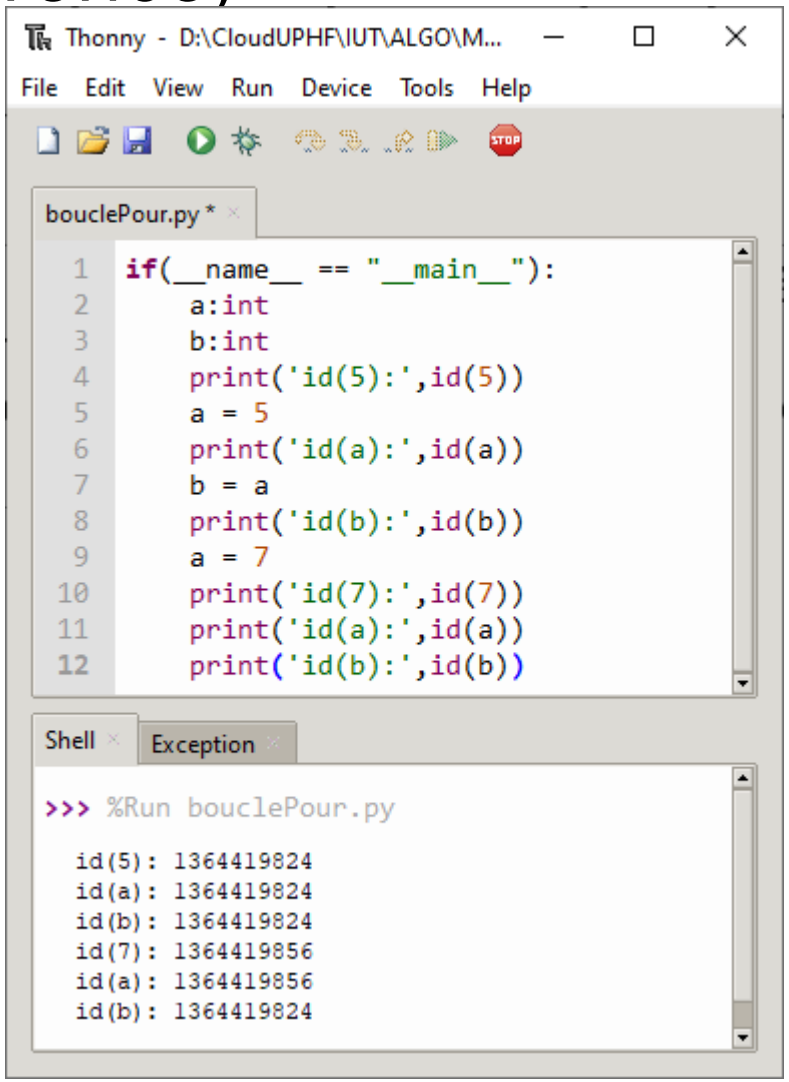
CHAPITRE 4

Identité des valeurs (référence)

Python associe à chaque valeur une référence, cette référence est généralement l'adresse mémoire où est rangée cette valeur.
La fonction **id()**, en python permet de récupérer cette référence.

Dans l'exemple on voit bien qu'au début **5**, **a** et **b** ont la même référence : **1364419824**
Puis on affecte à **a** la valeur **7**, on a alors **7** et **a** qui ont la même référence: **1364419856**

Au passage, bien qu'avant **a** et **b** avaient la même référence, le fait de modifier **a** n'a pas modifié **b**.



Données composites?

Par définition, données constituées de plusieurs éléments. **Une chaîne de caractères** (en Python, **str**) est une structure de données composée de plusieurs caractères.
En Python, il existe d'autres structures composites très utilisées :

- Les **listes**: structure **modifiable** permettant de stocker une suite d'éléments.
- Les **tuples**: structure **non modifiable** permettant de stocker une suite d'éléments.
- Les **dictionnaires**: structure permettant de stocker une suite d'éléments et d'y **accéder par une clé**.

Les listes

Une liste est une suite d'éléments qui se suivent. On décrit les éléments d'une liste entre crochets, séparés par des virgules :

maListe = [2, 3, 5, 7, 11]

Attention une liste est un objet modifiable!!!!

La méthode **append** permet d'ajouter un élément à une liste:

```
suite = [13, 17, 19]
maListe.append(suite)
print(maListe)
```

>>[2,3,5,7,11,[13,17,19]]

On remarquera qu'on n'a pas ajouté 3 valeurs, mais une liste contenant 3 valeurs!

Références et listes

```
[1]: a = [1,2,3]
     b=a
     print('a=',a,'id(a)=',id(a))
     print('b=',b,'id(b)=',id(b))

a= [1, 2, 3] id(a)= 100021632
b= [1, 2, 3] id(b)= 100021632
```

```
[2]: a.append(12)
     print('a=',a,'id(a)=',id(a))
     print('b=',b,'id(b)=',id(b))

a= [1, 2, 3, 12] id(a)= 100021632
b= [1, 2, 3, 12] id(b)= 100021632
```

```
[3]: c=a.copy()
     print('a=',a,'id(a)=',id(a))
     print('b=',b,'id(b)=',id(b))
     print('c=',c,'id(c)=',id(c))

a= [1, 2, 3, 12] id(a)= 100021632
b= [1, 2, 3, 12] id(b)= 100021632
c= [1, 2, 3, 12] id(c)= 100022312
```

```
[4]: b.append(15)
     print('a=',a,'id(a)=',id(a))
     print('b=',b,'id(b)=',id(b))
     print('c=',c,'id(c)=',id(c))

a= [1, 2, 3, 12, 15] id(a)= 100021632
b= [1, 2, 3, 12, 15] id(b)= 100021632
c= [1, 2, 3, 12] id(c)= 100022312
```

a et **b** ont la même référence vers le même objet **list**, ajouter des éléments dans l'objet **list** ne modifie pas sa référence. Ainsi on peut modifier la liste en utilisant indifféremment **a** ou **b**.
Pour créer une copie d'un objet **list** on utilise la méthode **copy()**.

c est une copie de **a** : quand on modifie, par la suite, l'objet **list** référencé par **a** on ne modifie pas l'objet **list** référencé par **c**.

Quelques opérations sur les listes

Pour créer une liste vide:

[1]:
```
suite = []
print('suite:',suite)
```

```
suite: []
```

Un objet de type **list** est modifiable
ajouter en fin de liste une valeur:

[2]:
```
suite.append(5)
print('suite:',suite)
```

```
suite: [5]
```

[3]:
```
autre = [7,9]
suite.append(autre)
print('suite:',suite)
```

```
suite: [5, [7, 9]]
```

[4]:
```
suite2=suite+[56]
print(suite)
print(suite2)
```

```
[5, [7, 9]]
[5, [7, 9], 56]
```

l'operateur + retourne une nouvelle liste correspondant à la concatenation des deux listes. Cette operation ne modifie aucune des deux listes

Les tuples

Un ***tuple*** est une suite d'éléments qui se suivent. On décrit les éléments d'un ***tuple*** entre parenthèses (parfois elles peuvent être omises), séparés par des virgules :

monTuple = (2, 3, 5, 7, 11)

Attention un ***tuple*** est un objet **non modifiable**!!!!

Création simple de tuples

Creation d'un tuple :

[1]:
```
t1 = 1,'toto',False
print("t1 = ",t1)
print(type(t1))
```

```
t1 =  (1, 'toto', False)
<class 'tuple'>
```

[2]:
```
t2 = ()
print("t2 = ",t2)
print(type(t2))
```

```
t2 =  ()
<class 'tuple'>
```

[3]:
```
t3 = (2)
print("t3 = ",t3)
print(type(t3))
```

```
t3 =  2
<class 'int'>
```

[4]:
```
t4  = (2,)
print("t4 = ",t4)
print(type(t4))
```

```
t4 =  (2,)
<class 'tuple'>
```

ATTENTION A LA SYNTAXE POUR CRÉER UN TUPLE D'UN SEUL ELEMENT : il faut mettre une virgule après le premier élément.

Les dictionnaires

Un dictionnaire (***dict***) est une collection d'éléments dont la valeur est associée à une clé (***key***). On décrit les éléments d'un dictionnaire entre accolades, on donne la clé : la valeur. Les couples clé:valeur sont séparés par des virgules :

monDict = {'nom':'Durant','prenom':'Jean','age':25}

Quelques opérations sur les *dict*

Céer un dictionnaire vide

[1]:
```
d1={}
```

ajouter un élement

[2]:
```
print(d1)
d1['x']=5
print(d1)
```

```
{}
{'x': 5}
```

Comme les **list** les **dict** sont muables il faut faire attention aux copies:

[3]:
```
d2 = d1
d3 = d1.copy()
d1['y'] = 3
print('d1=',d1, 'id(d1)=', id(d1))
print('d2=',d2, 'id(d2)=', id(d2))
print('d3=',d3, 'id(d3)=', id(d3))
```

```
d1= {'x': 5, 'y': 3} id(d1)= 90421888
d2= {'x': 5, 'y': 3} id(d2)= 90421888
d3= {'x': 5} id(d3)= 92850624
```

[1]:
```
str1 = 'hello world'
list1 = ['un',2, 3.0, [54,45]]
tuple1 = ('abc', 23, 56)
dict1 = {'nom':'caisse1','largeur':10,'longueur':12,'hauteur':5}
```

connaître le nombre d'éléments: la fonction **len**

[2]:
```
print('taille de str1:',len(str1))
print('taille de list1:',len(list1))
print('taille de tuple1:',len(tuple1))
print('taille de dict1:',len(dict1))
```

```
taille de str1: 11
taille de list1: 4
taille de tuple1: 3
taille de dict1: 4
```

accéder à l'ième élément en partant du début(sauf dictionnaire): l'opérateur []

[3]:
```
i = 2
print('str1[',i,']:',str1[i])
print('list1[',i,']:',list1[i])
print('tuple1[',i,']:',tuple1[i])
print("dict1['largeur']=",dict1['largeur'])
```

```
str1[ 2 ]: l
list1[ 2 ]: 3.0
tuple1[ 2 ]: 56
dict1['largeur']= 10
```

accéder à l'ième élément en partant de la fin(sauf dictionnaire): l'opérateur []

[4]:
```
i = 1
print('str1[',-i,']:',str1[-i])
print('list1[',-i,']:',list1[-i])
print('tuple1[',-i,']:',tuple1[-i])
```

```
str1[ -1 ]: d
list1[ -1 ]: [54, 45]
tuple1[ -1 ]: 56
```

Parcours de données composites (1)

parcours des éléments d'une chaîne de caractères (str)

[5]:
```
for x in str1 :
    print('x=',x)
```

```
x= h
x= e
x= l
x= l
x= o
x=
x= w
x= o
x= r
x= l
x= d
```

Parcours de données composites (2)

parcours des éléments d'une liste (list)

[6]:
```
for x in list1 :
    print('x=',x)
```

```
x= un
x= 2
x= 3.0
x= [54, 45]
```

Parcours de données composites (3)

parcours des éléments d'un tuple

[7]:
```
for x in tuple1 :
    print('x=',x)
```

```
x= abc
x= 23
x= 56
```

Parcours de données composites (4)

parcours des éléments d'un dictionnaire (dict)

[8]:
```
for x in dict1:
    print('x=',x)
```

```
x= nom
x= largeur
x= longueur
x= hauteur
```

[9]:
```
for x in dict1.values() :
    print('x=',x)
```

```
x= caisse1
x= 10
x= 12
x= 5
```

[10]:
```
for x in dict1.items() :
    print('x=',x)
```

```
x= ('nom', 'caisse1')
x= ('largeur', 10)
x= ('longueur', 12)
x= ('hauteur', 5)
```

déballage

Consiste a affecter a des variables les valeurs d'une donnée composites. Quelques exemples :

[11]:
```
a,b,c,d,e,f,g,h,i,j,k = str1
print('a=',a, 'c=', c, 'k=',k)
```

```
a= h c= l k= d
```

[12]:
```
v1,v2,v3,v4 = list1
print('v1=',v1,'v2=',v2, 'v3=',v3,'v4=',v4)
```

```
v1= un v2= 2 v3= 3.0 v4= [54, 45]
```

[13]:
```
p1,p2,p3 = tuple1
print("p1=",p1,"p2=",p2,"p3=",p3)
```

```
p1= abc p2= 23 p3= 56
```

[16]:
```
un, deux, trois, quatre = dict1
print("un=",un,"deux=",deux,"trois=",trois,"quatre=",quatre)
```

```
un= nom deux= largeur trois= longueur quatre= hauteur
```

Déballage : le cas des dictionnaires (*dict*)

[14]:
```
un, deux, trois, quatre = dict1.items()
print("un=",un,"deux=",deux,"trois=",trois,"quatre=",quatre)
```

```
un= ('nom', 'caisse1') deux= ('largeur', 10) trois= ('longueur', 12) quatre=
('hauteur', 5)
```

[15]:
```
x,y = un
print("x=",x,"y=",y)
```

```
x= nom y= caisse1
```

[16]:
```
for cle,valeur in dict1.items():
    print("cle =",cle,"valeur=",valeur)
```

```
cle = nom valeur= caisse1
cle = largeur valeur= 10
cle = longueur valeur= 12
cle = hauteur valeur= 5
```

Suppression d'élément d'une liste (*list*)

la méthode **remove()** :

```
In [8]: a = ['un','deux','un','trois']
        a.remove('un')
        print(a)

['deux', 'un', 'trois']
```

la fonction **del()** :

```
In [10]: a = ['un','deux','un','trois']
         del(a[1])
         print(a)

['un', 'un', 'trois']
```

la méthode **pop()**

```
In [12]: a = ['un','deux','un','trois']
         x = a.pop(1)
         print(a)
         print(x)

['un', 'un', 'trois']
deux
```

Suppression d'un élément d'un dictionnaire (*dict*)

la fonction **del()** :

```
In [16]: placard = {"chemise":3, "pantalon":6, "tee shirt":7}
         del(placard["chemise"])
         print(placard)

{'pantalon': 6, 'tee shirt': 7}
```

la méthode **pop()** :

```
In [17]: placard = {"chemise":3, "pantalon":6, "tee shirt":7}
         x = placard.pop("chemise")
         print(placard)
         print(x)

{'pantalon': 6, 'tee shirt': 7}
3
```

Université Polytechnique HAUTS-DE-FRANCE

les fonctions

CHAPITRE 5

Fonction : notion de contexte

Quand une fonction est exécutée un contexte lui est associé:
Un contexte est une zone mémoire dans laquelle vont être stockées:

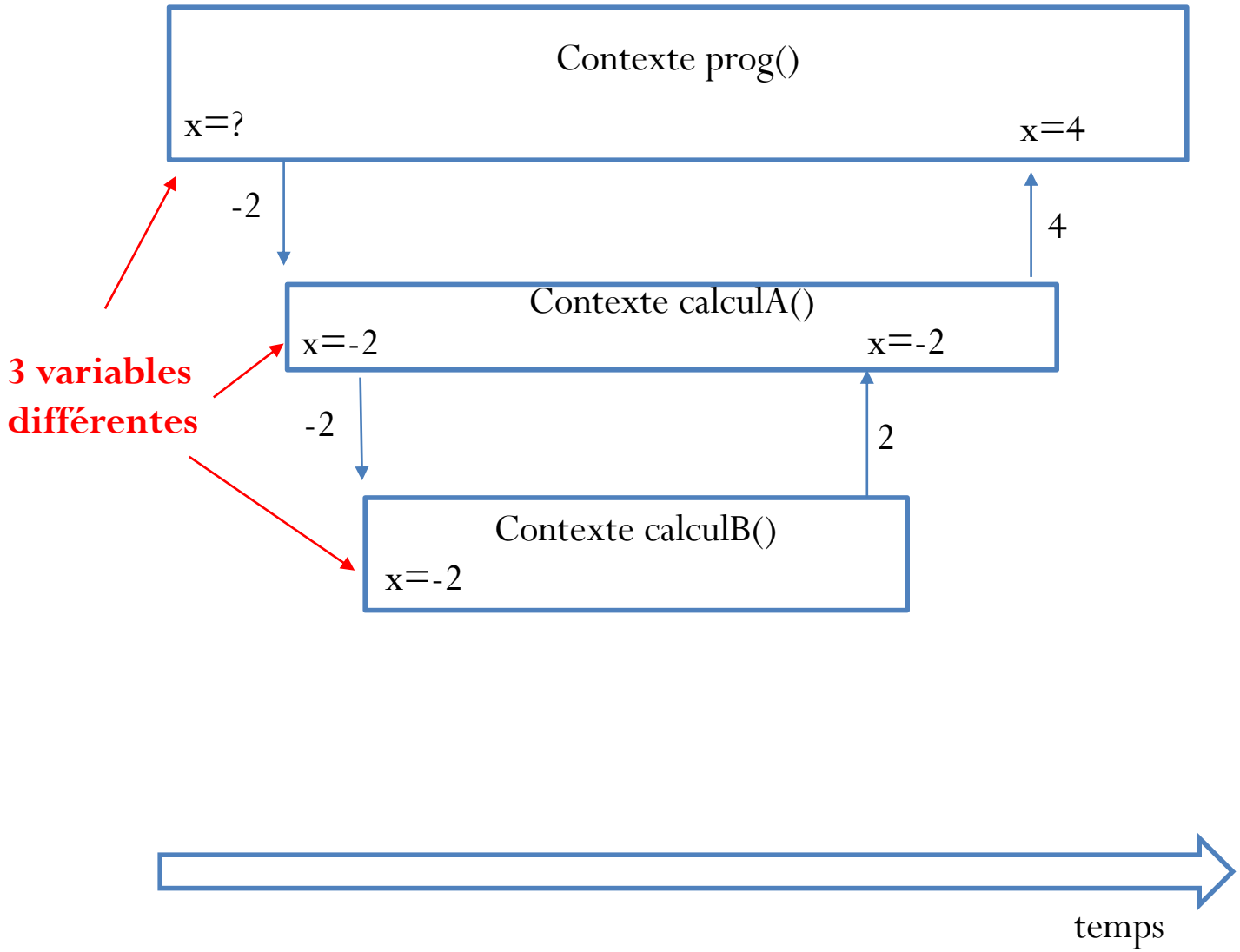
- la valeur des paramètres
- la valeur des variables

Tant que la fonction n'est pas terminée, le contexte reste chargé en mémoire.

Les variables dans les fonctions appartiennent au contexte

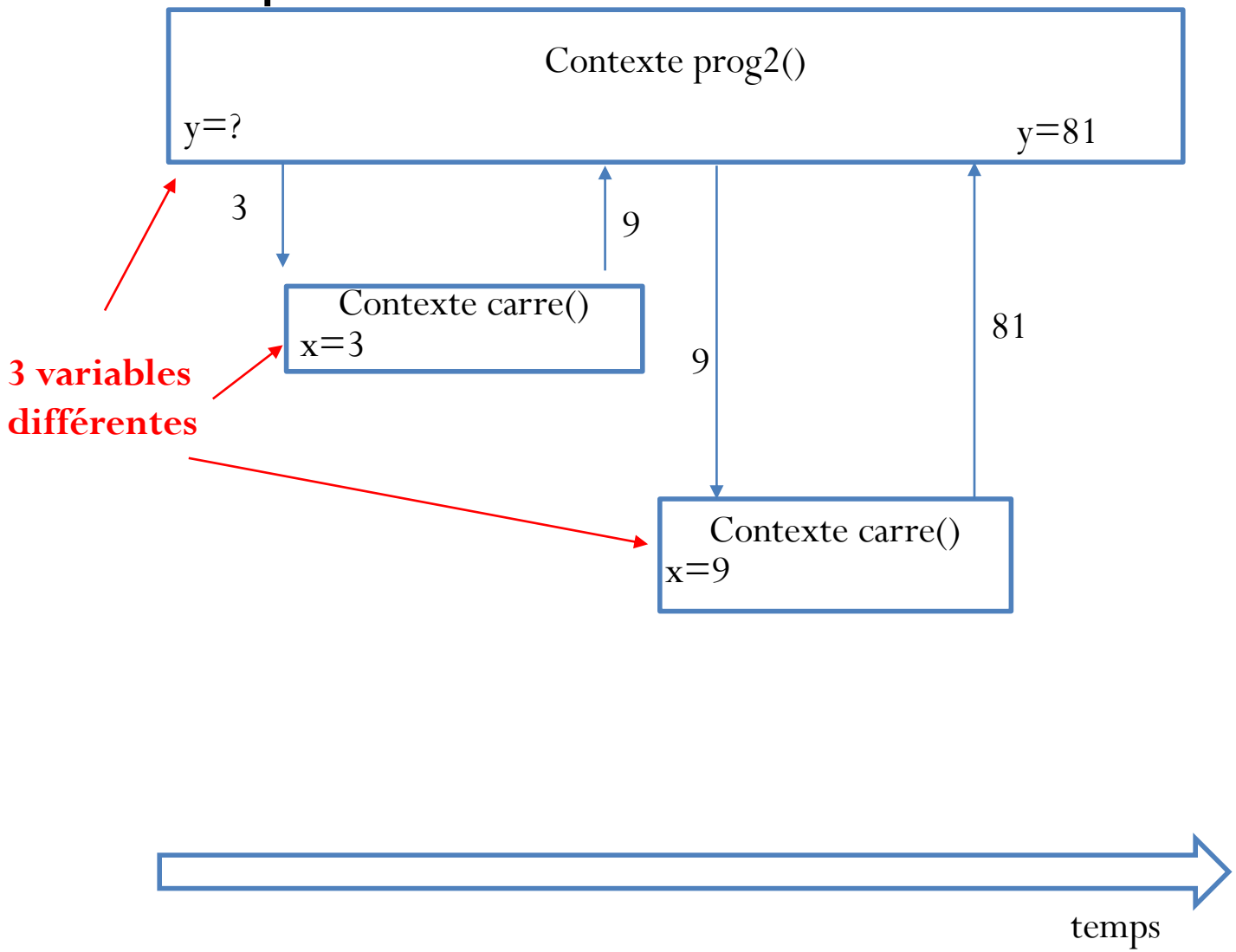
Exemple 1:

```
def calculB(x:int)->int:
    if(x<0):
        return -x
    else:
        return x


def calculA(x:int)->int:
    return 2*calculB(x)


def prog():
    x = calculA(-2)
    print("x=",x)


if(__name__=="__main__"):
    prog()
```



Exemple 2:

```
def carre(x:float)->float:
    return x*x

def prog2():
    y=carre(carre(3))
    print("y= ",y)


if(__name__=="__main__"):
    prog2()
```



Qu'affiche ce programme?

```
def permut(a:int,b:int)->None:
    tmp:int
    tmp = a
    a = b
    b = tmp
    print("dans permut a = ", a, " b= ",b)

def prog3():
    a:int
    b:int
    a = 5
    b = 10
    permut(a,b)
    print("dans prog3 a = ", a, " b= ",b)

if(__name__=="__main__"):
    prog3()
```

Réponse :

[18]:
```
def permut(a:int,b:int)->None:
    tmp:int
    tmp = a
    a = b
    b = tmp
    print("dans permut a = ", a, " b= ",b)

def prog3():
    a:int
    b:int
    a = 5
    b = 10
    permut(a,b)
    print("dans prog3 a = ", a, " b= ",b)

if(__name__=="__main__"):
    prog3()
```

```
dans permut a =  10  b=  5
dans prog3 a =  5  b=  10
```

La récursivité:

- Algorithme d'une fonction calculant la somme des *n* premiers entiers:
- Soit S_n cette somme,
- $S_n = 1 + 2 + \cdots + n - 1 + n$
- $S_n = n + S_{n-1}$

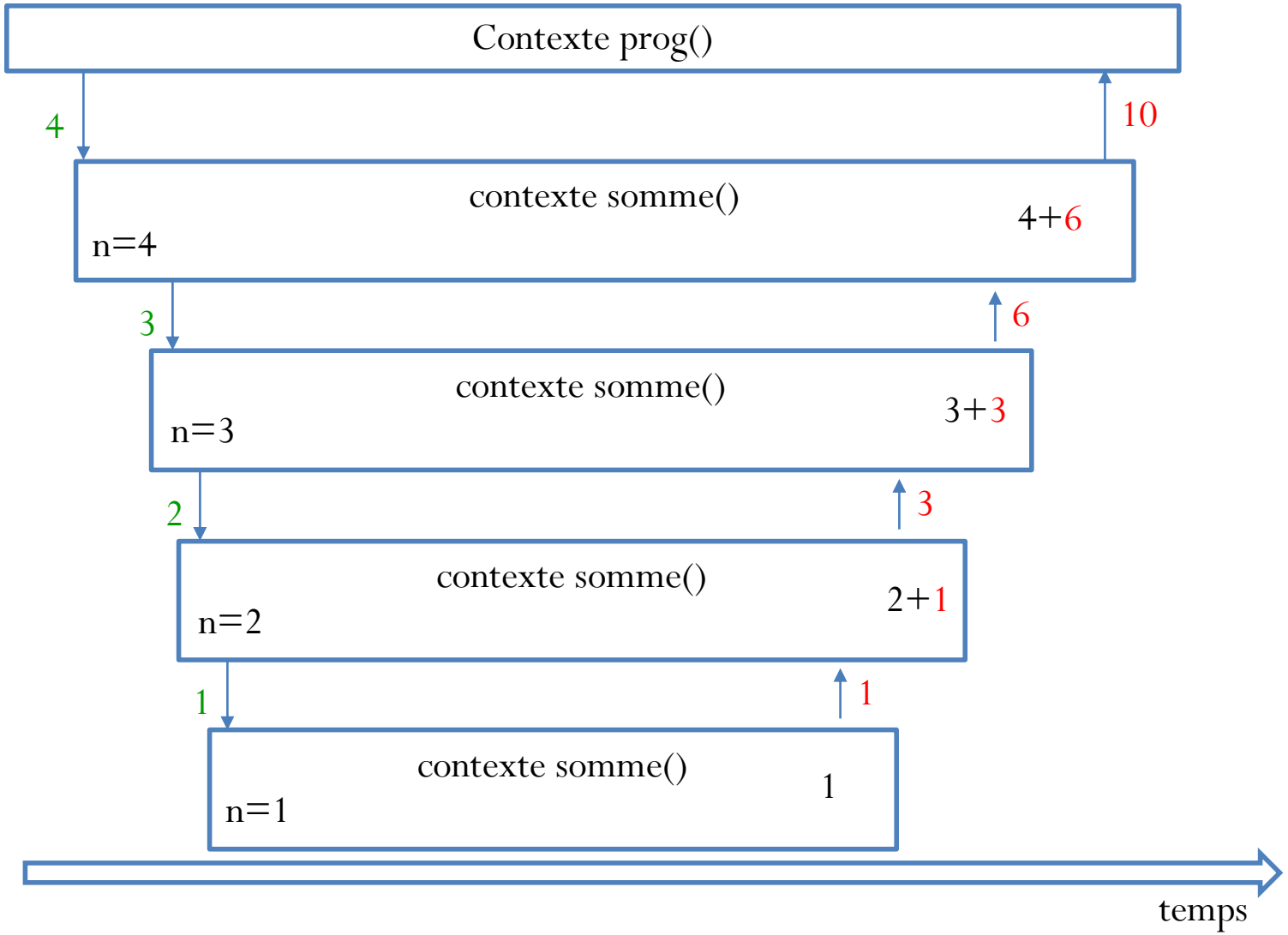
$$S_1 = 1$$

Algorithme 13 somme récursive

```
1: fonction SOMME(n : entier) : entier
2:     Si n > 1 alors
3:         renvoyer n + somme(n − 1)
4:     Sinon
5:         renvoyer 1
6:     Fin si
7: Fin fonction
```

Code Python

```
def somme(n:int)->int:
    if(n>1):
        return n+somme(n-1)
    else:
        return n

def prog():
    print(somme(4))

if(__name__=="__main__"):
    prog()
```



La récursivité vs itération

La plupart des algorithmes comportant des répétitions peuvent se traduire sous une forme récursive ou itérative (boucle pour, tant que…).
Selon la nature du problème la solution sera naturellement récursive ou itérative.
L'exécution d'un code récursif nécessite d 'empiler les contextes des appels des fonctions ce qui peut être limitatif dans certains langages notamment en Python

Récursivité terminale

```
def sommeTerminale(n:int,res:int)->int:
    if(n>1):
        return sommeTerminale(n-1,n+res)
    else:
        return res+n

def somme2(n:int)->int:
    return sommeTerminale(n,0)

def prog():
    print(somme2(4))

if(__name__=="__main__"):
    prog()
```

Principe :

Le résultat intermédiaire est transmis lors de l'appel récursif

Pas de calcul après l'appel, lors du retour.

Avantage :

La plupart des compilateurs transforme la récursivité terminale en traitement itératif!

→ Avantage pour le développeur qui peut « penser récursif », et amélioration des performances. Pas de problème de limitation de pile d'appel!

→ Pas réalisé en python

Paramètres de fonction par défaut

Il est possible en Python de donner une valeur aux derniers paramètres d'une fonction, par exemple:

[19]:
```
def boucle(start:int,fin:int=10,pas:int=1)->None:
    cpt:int
    cpt=start
    while(cpt<=fin):
        print(cpt)
        cpt=cpt+pas

boucle(2,13,3)
```

```
2
5
8
11
```

[20]:
```
boucle(1,5)
```

```
1
2
3
4
5
```

[21]:
```
boucle(3)
```

```
3
4
5
6
7
8
9
10
```

Université Polytechnique HAUTS-DE-FRANCE

Algorithmes de tri

CHAPITRE 6

Trier les valeurs d'un vecteur

- Trier c'est ranger dans un certain ordre.
- Il faut donc un critère de comparaison.
- Différents algorithmes ont été proposés pour trier des valeurs dans un vecteur, chaque algorithme a sa performance, qu'on exprime selon sa complexité (ordre de grandeur du nombre d'opérations nécessaire pour réaliser le traitement). Généralement, l'ordre de grandeur s'exprime en fonction du nombre d 'éléments (taille) du vecteur.

Tester si une liste est triée

```
def isSorted(tab:list)->bool:
    """
        retourne True si la list tab est triée dans l'ordre croissant
    """
    ok:bool # booleen passant à faux si deux valeurs ne sont pas ordonnées
    i:int # compteur de boucle pour avancer dans la liste
    ok=True
    i = 1
    while i<len(tab) and ok :
        if(tab[i-1]<=tab[i]):
            # on avance dans le tableau
            i = i+1
        else:
            #on sait que tab n'est pas trié
            ok = False
    return ok
```

Le tri par insertion

Source de l'animation: wikipédia

6 5 3 1 8 7 2 4

Code python du tri par insertion

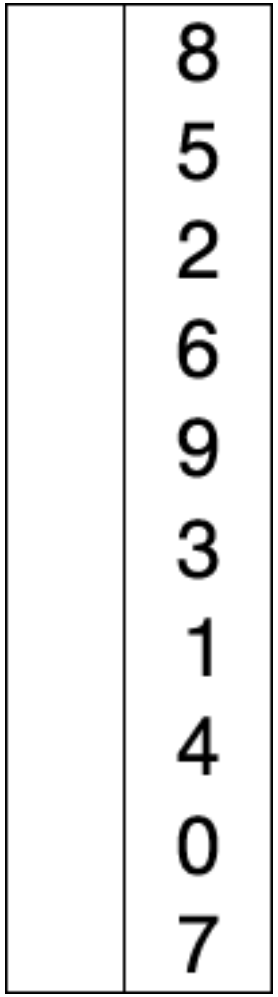
```
def triInsertion(t:list)->None :
    i:int
    j:int
    p:int

    for i in range(1,len(t)):
        p = 0
        while p<i and t[i]>t[p]:
            p=p+1
        if(p!=i):
            tmp = t[i]
            for j in range(i,p,-1):
                t[j] = t[j-1]
            t[p] = tmp
```

Explication du principe

- Cf video sur moodle

Tri par sélection

- Source de l'animation: wikipédia



Principe du tri par sélection

- On considère 2 partitions : 1 triée et 1 non triée
- Au démarrage la partition triée est vide et la partition non triée est composée de tous les éléments de la liste
- À chaque itération de l'algorithme :
  - On augmente d'une case la partition triée (la première valeur de la partition non triée devient la dernière valeur de la partition triée)
  - On recherche dans la partition non triée la plus petite valeur
  - Si cette valeur est plus petite que la dernière de la partition triée on les permute.

illustration

On veut trier [3,6,4,1,7,2]

Itération 1 (en vert partition triée, en rouge partition non triée):
- on va chercher quelle valeur placer en position 0
- [3,6,4,1,7,2],on cherche l'indice de la valeur minimale dans la partition non triée => 3
- Comme 3 ≠ 0, on permute les valeurs des positions 0 et 3

Itération 2 (en vert partition triée, en rouge partition non triée):
- on va chercher quelle valeur placer en position 1
- [1,6,4,3,7,2],on cherche l'indice de la valeur minimale dans la partition non triée => 5
- Comme 5 ≠ 1, on permute les valeurs des positions 1 et 5

Itération 3 (en vert partition triée, en rouge partition non triée):
- on va chercher quelle valeur placer en position 2
- [1,2,4,3,7,6],on cherche l'indice de la valeur minimale dans la partition non triée => 3
- Comme 2 ≠ 3, on permute les valeurs des positions 2 et 3

Itération 4 (en vert partition triée, en rouge partition non triée):
- on va chercher quelle valeur placer en position 3
- [1,2,3,4,7,6],on cherche l'indice de la valeur minimale dans la partition non triée => 3

Itération 5 (en vert partition triée, en rouge partition non triée):
- on va chercher quelle valeur placer en position 4
- [1,2,3,4,7,6],on cherche l'indice de la valeur minimale dans la partition non triée => 5
- Comme 4 ≠ 5, on permute les valeurs des positions 4 et 5
- [1,2,3,4,6,7],

```
def triSelection(tab:list)->None:
    i:int
    j:int
    imin:int
    for i in range(0,len(tab)-1):
        imin = i
        for j in range(i+1,len(tab)):
            if (tab[j]<tab[imin]):
                imin = j
        if(imin!=i):
            tmp = tab[i]
            tab[i] = tab[imin]
            tab[imin] = tmp
```

Université Polytechnique HAUTS-DE-FRANCE

Créer des listes et des matrices

CHAPITRE 7

Créer une liste possédant « n cases »

- On souhaite créer un liste possédant un nombre défini de cases, initialisées chacune par une valeur fixée:

```
def construireListe(taille:int, valeur:float) -> list :
    res = []
    for i in range(0,taille):
        res.append(valeur)
    return res
```

Une matrice?

- Une matrice est un « tableau » de ***n* lignes** et de ***m* colonnes**
- **Exemple :**

3	5.5	7.2	9
32	55	72	9
321	532	241	2

- Une matrice de 3 lignes et 4 colonnes peut être considérée comme une liste de 3 éléments dont chaque élément est une de 4 éléments (ici des réels)

Construire une matrice de n lignes m colonnes?

```
def contruireMatrice(nbLignes:int, nbColonnes:int, valeur:float)->list:
    res = []
    for i in range(0,nbLignes):
        lig = []
        for j in range(0,nbColonnes):
            lig.append(valeur)
        res.append(lig)
    return res
```

Accéder à un élément de la matrice:

Soit :

toto = construireMatrice(2,3,1)

Que vaut toto[0]?

toto est une list de 2 éléments, chacun étant une list de 3 réels.

toto[0] est dont la première list : [1,1,1]

Que vaut toto[1][2]?

toto[1] : c'est la dernière ligne représentée par une list : [1,1,1], donc toto[1][2] est donc le dernier élément de la dernière ligne soit ici 1

Erreur liée aux références de list

```
def construireMatriceV2(nbLignes:int, nbColonnes:int, valeur:float)->list:
    res = []
    lig = []
    for j in range(0,nbColonnes):
        lig.append(valeur)
    for i in range(0,nbLignes):
        res.append(lig)

    return res

mat = construireMatriceV2(2,3,1)
mat[0][0] = 9
print(mat)
```

```
>> [[9, 1, 1], [9, 1, 1]]
```

Utilisation de motif à répéter pour créer une liste

Notation :

[motif] * nombre de fois à répéter

Exemple :

- `>>> [1,[2]]*3`
- `[1, [2], 1, [2], 1, [2]]`

Application

- Création d'une liste de n éléments:

```
def buildList(n:int)->list:
    return [None]*n
```

```
>>> buildList(3)
[None, None, None]
>>> buildMat(2,4)
[[None, None, None, None], [None, None, None, None]]
>>>
```

- Création d'une matrice de n lignes m colonnes: (**même erreur que l'exemple construireMatriceV2**)

```
def buildMat(n:int,m:int)->list:
    return [[None]*m]*n
```

On recopie n fois la même ligne

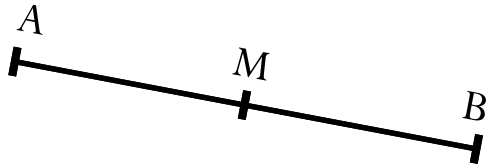
```
def buildMat(row:int, column:int) ->list :
    mat = []
    for lig in range(0,row):
        mat.append([None]*column)
    return mat
```

Créer sa structure de données dynamique

CHAPITRE 8

Exemple de problème : calcul des coordonnées du milieu d'un segment

Soit M: $milieu\ de\ [AB] \Rightarrow \begin{cases} x_M = \frac{x_A - x_B}{2} \\ y_M = \frac{y_A - y_B}{2} \end{cases}$



Comment écrire une fonction qui retourne 2 valeurs ($x_M et\ y_M$) ?

➔ Ce n'est pas possible, une fonction ne peut pas retourner plusieurs valeurs

➔ Alternative 1 : utiliser des pointeurs ➔ alourdit l'appel de fonctions, ne permet pas d'enchaîner des appels

➔ Alternative 2 : utiliser des tableaux ➔ seulement possible si les valeurs sont de même nature

Les types personnalisés

Il est souvent utile de pouvoir créer ses propres types. Il s'agit alors de définir une structure permettant de regrouper plusieurs sous-valeurs pouvant être de nature différentes (appelés champs).

Par exemple, on veut représenter un point, chaque sera décrit par :

- Son abscisse (réel)
- Son ordonnée (réel).

Le type Point sera un type structuré par 2 champs

Définitions

Nous avons vu les collections, notamment les ***list***, qui sont des types permettant de regrouper sous un même identificateur plusieurs données.

Type complexe hétérogène : type de données décrivant une information composite, constituée de plusieurs valeurs qui peuvent être elles-mêmes de types différents (qui peuvent être eux-mêmes simples ou complexes). Autrement dit, une structure permet de regrouper sous un même identificateur plusieurs données qui peuvent être complètement différentes. Ces données sont appelées les champs de la structure.

Champ de structure : élément constituant la structure, caractérisé par un identificateur et un type. Si on considère une structure comme un groupe de variables, alors un champ peut être comparé à une variable. En effet, comme une variable, un champ possède un type, un identificateur, une valeur et une adresse. Cependant, à la différence d'une variable, un champ n'est pas indépendant, car il appartient à un groupe de champs qui forme sa structure. Le fait d'associer un identificateur à un champ est important, car cela signifie que chaque élément d'une structure possède un nom spécifique. Ce nom est unique dans la structure, et permet d'identifier le champ sans ambiguïté. Par opposition, les éléments qui constituent un tableau ne possèdent pas de nom : ils sont simplement numérotés, grâce à leur index qui indique leur position dans le tableau. Un champ peut être de n'importe quel type, qu'il soit simple ou complexe. Il peut donc s'agit d'un tableau, ou même d'une structure.

Types complexes hétérogènes vus :

- list
- dict
- tuple

INCONVENIENT :
impossible d'imposer le nombre et le type des éléments

Solution

- les namedtuple
- les classes (programmation objet)

Exemple de creation d’un namedtuple

[1]:
```
from collections import namedtuple
pixel = namedtuple("Point",["x","y"])
p1 = pixel(1,3)
print(p1)
print(type(p1))
print(p1.y)
```

```
Point(x=1, y=3)
<class '__main__.Point'>
3
```

[2]:
```
p1.x = 7
```

```
      ␣
↪-------------------------------------------

      AttributeError
↪last)

      <ipython-input-2-59d3e7cc377e> in <modul
   ----> 1 p1.x = 7


      AttributeError: can't set attribute
```

importer namedtuple de la bibliotheque collections

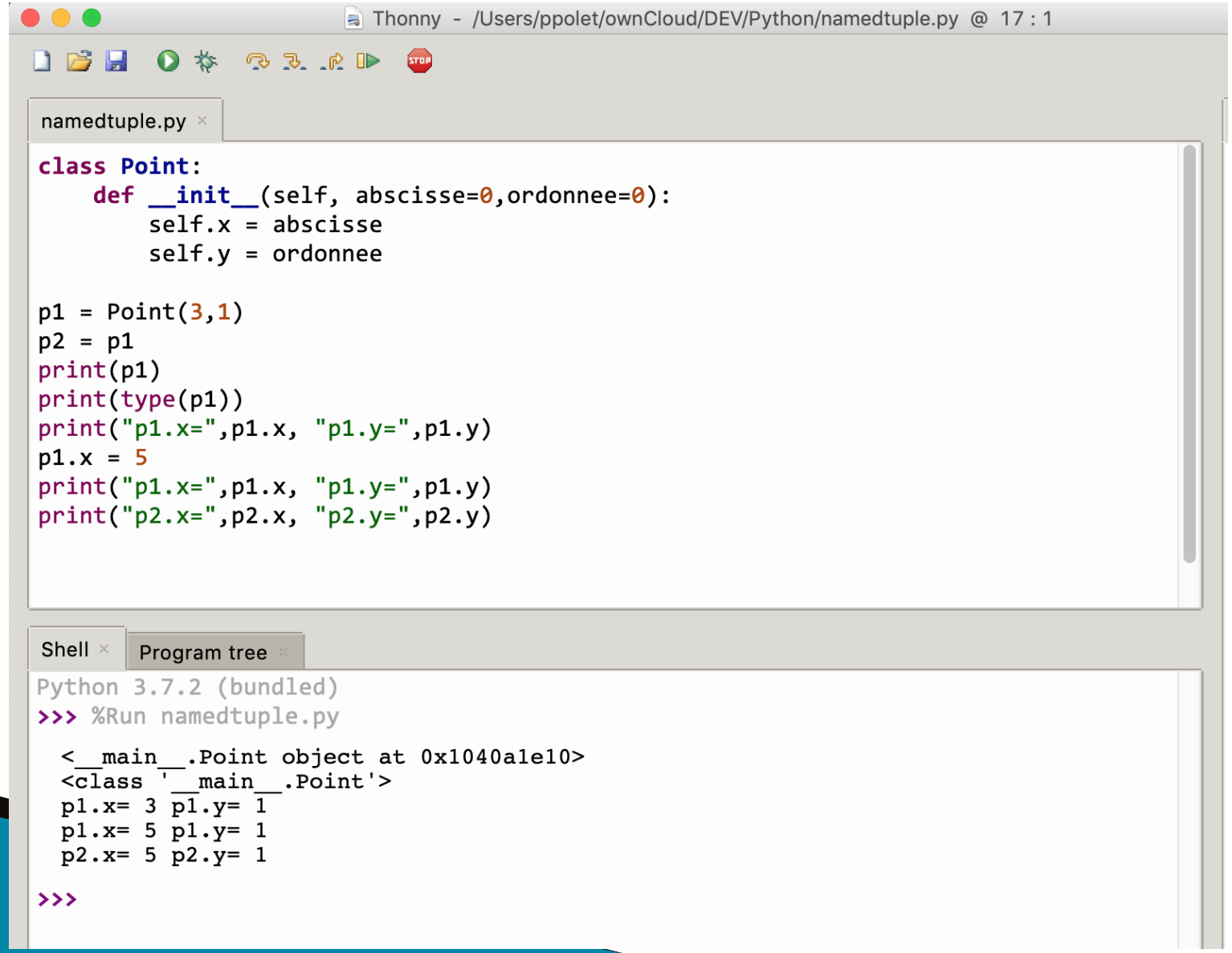
2 paramètres:

- le nom du type
- une liste de noms de champs

accès à un des champs

ATTENTION modification impossible

Utiliser les classes pour faire des types structurés



Toujours le même nom

__init__ : méthode décrivant comment initialiser les attributs à la création

p1 contient « la référence » d'un objet Point

p2 = p1 → p2 et p1 référencent le même objet

créer une copie d'un objet:

```
class Point:
    def __init__(self, abscisse=0,ordonnee=0):
        self.x = abscisse
        self.y = ordonnee

def copier(p:Point)->Point:
    return Point(p.x,p.y)

p1 = Point(3,1)
p2 = p1
p3=copier(p1)
print(p1)
print(type(p1))
print("p1.x=",p1.x, "p1.y=",p1.y)
p1.x = 5
print("p1.x=",p1.x, "p1.y=",p1.y)
print("p2.x=",p2.x, "p2.y=",p2.y)
print("p3.x=",p3.x, "p3.y=",p3.y)
```

Shell | Program tree

```
>>> %Run namedtuple.py

  <__main__.Point object at 0x10f75f400>
  <class '__main__.Point'>
  p1.x= 3 p1.y= 1
  p1.x= 5 p1.y= 1
  p2.x= 5 p2.y= 1
  p3.x= 3 p3.y= 1
```

Introduction aux listes chaînées

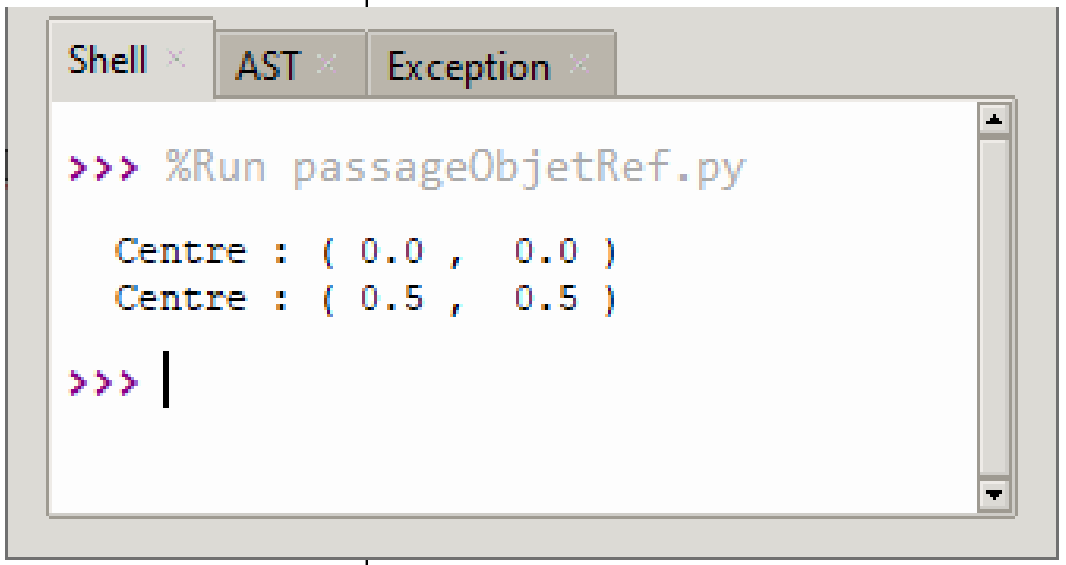
- Comment définir le type Liste?

- Notions de :
  - ensemble d'éléments
  - ordre (premier, dernier, suivant…) des éléments
  - accès à un élément
  - possibilité de modifications (ajout, suppression)…

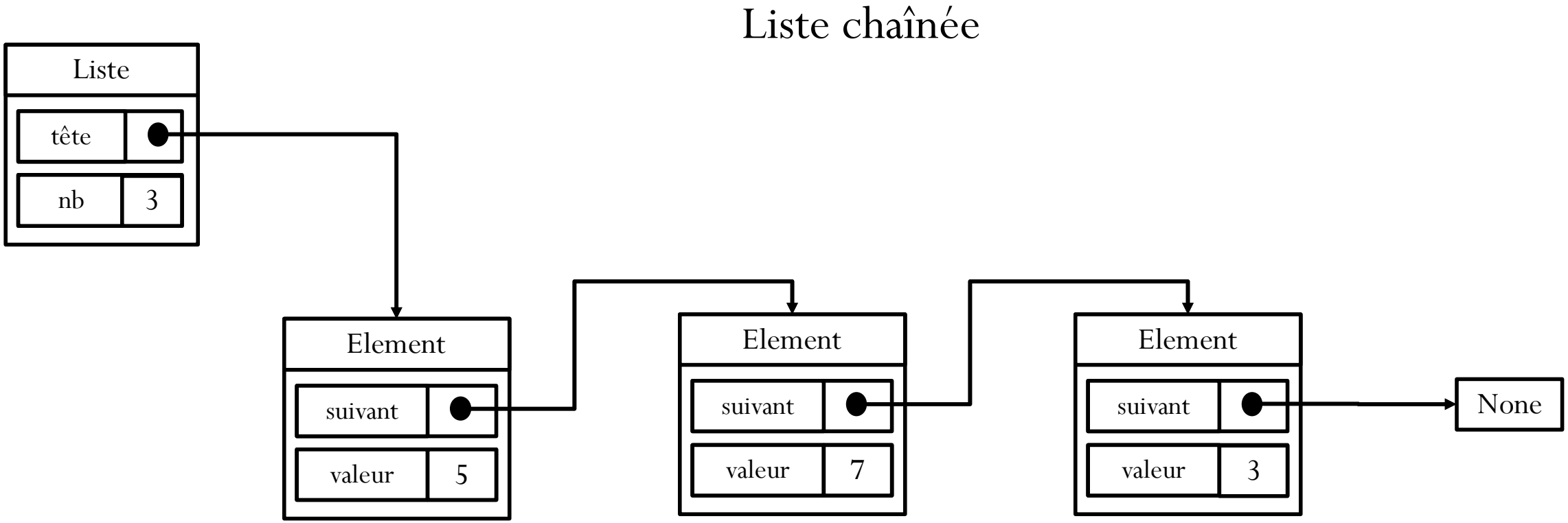
Implantation possible par une ***liste chaînée***

Rappels

```
class Point:
    def __init__(self,vx=0.0, vy=0.0):
        self.x=vx
        self.y=vy


def modifPoint(p:Point):
    p.x = p.x + 0.5
    p.y = p.y + 0.5

def prog():
    centre = Point()
    print("Centre : (",centre.x,", ",centre.y,")")
    modifPoint(centre)
    print("Centre : (",centre.x,", ",centre.y,")")

if __name__ == "__main__" :
    prog()
```

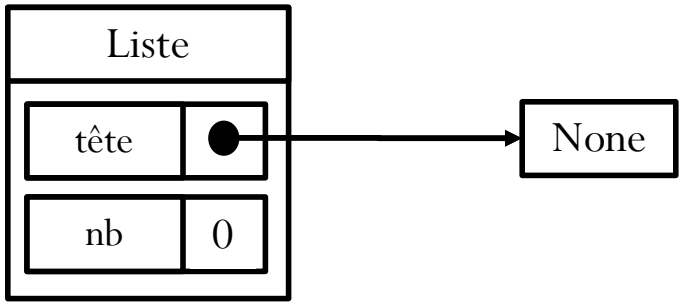


Représentation schématique (cas général)



Représentation schématique (cas particulier)

Liste chaînée vide



Définition des classes en Python

- La classe **Element**

```
class Element:
    def __init__(self,v):
        self.valeur = v
        self.suivant = None
```

La classe **Liste**

```
class Liste:
    def __init__(self):
        self.tete = None
        self.nb = 0
```

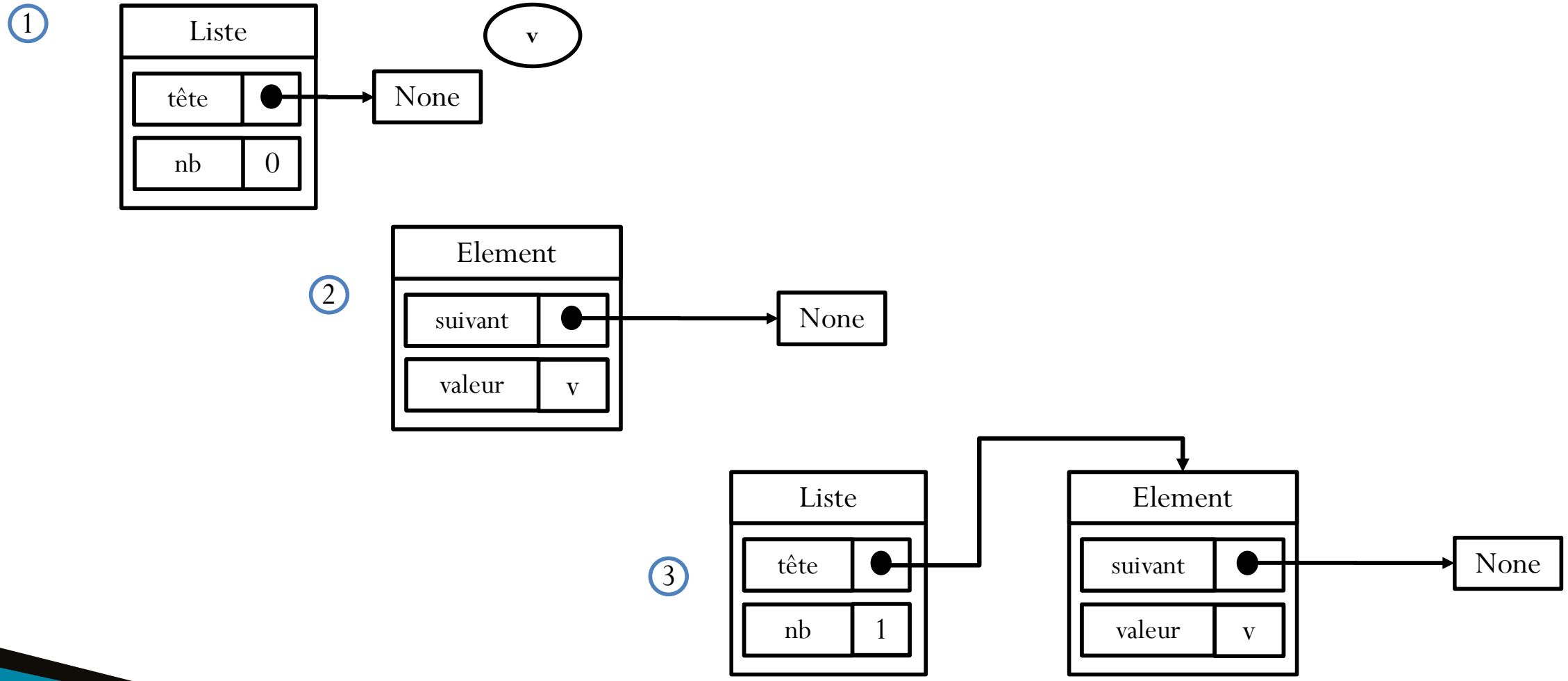
Modifier un ***Element***:

```
def setValeur(elt:Element, v):
    elt.valeur = v

def setSuivant(elt:Element, succ:Element):
    elt.suivant = succ
```

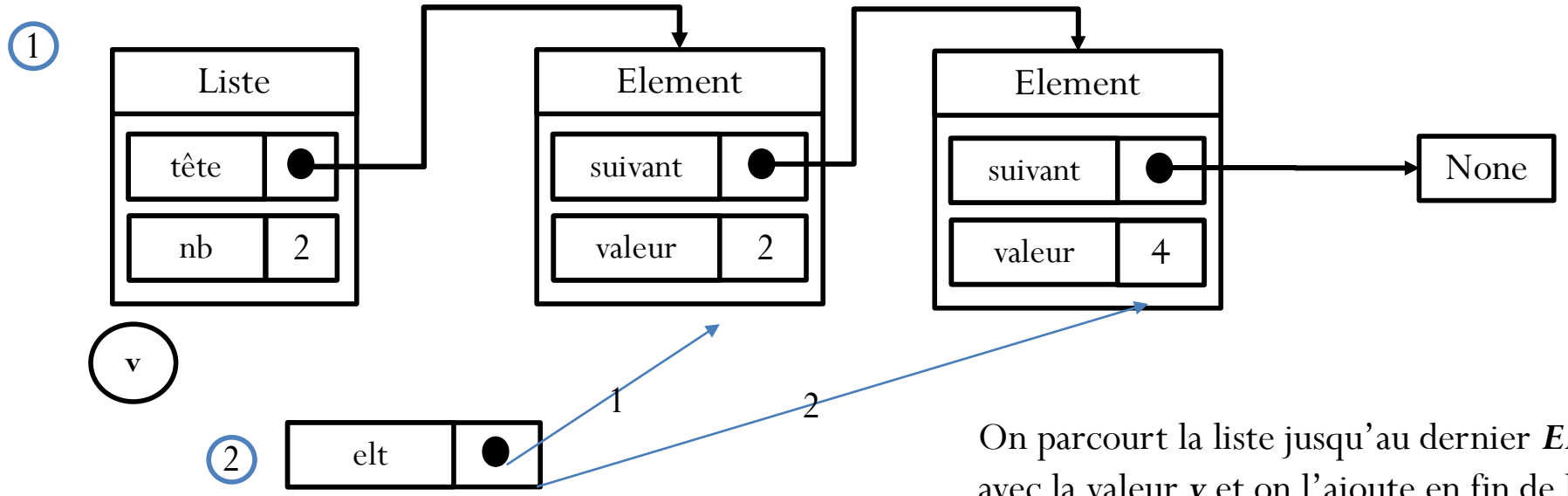
Afficher une ***Liste***

```
def afficherListe(lst:Liste):
    print("[ ",end="")
    elt = lst.tete
    while elt!=None :
        print(elt.valeur,end="")
        elt = elt.suivant
        if(elt!=None):
            print(", ",end="")
    print("]")
```

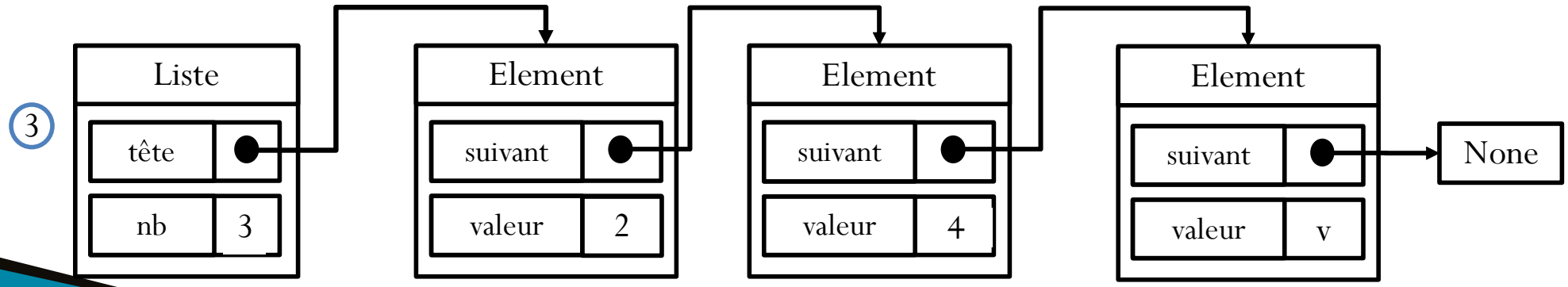
Ajouter un valeur dans une ***Liste*** vide



Ajouter un valeur dans une ***Liste***



On parcourt la liste jusqu'au dernier ***Element***, puis on crée un ***Element*** avec la valeur ***v*** et on l'ajoute en fin de la ***Liste***



Ajouter un valeur dans une *Liste*

```
def ajouterValeur(lst:Liste,v)->Liste:
    lst.nb = lst.nb+1
    if lst.tete == None:
        lst.tete=Element(v)
        return lst

    elt = lst.tete

    while elt.suivant!=None:
        elt = elt.suivant
    nouv = Element(v)
    elt.suivant = nouv
    return lst
```

Récupérer la i^{ème} valeur d'une ***Liste***

```
def getAt(lst:Liste,n:int):
    if(n>lst.nb) or n<1:
        return None
    cpt = 1
    elt = lst.tete
    while cpt<n:
        elt = elt.suivant
        cpt = cpt + 1
    return elt.valeur
```

La 1ère valeur est à la position 1

On pourrait lever une Exception (voir le cours consacré aux Exceptions)

Supprimer la i^{ème} valeur d'une ***Liste***

Principe:

Parcourir les ***Element*** de la ***Liste*** et s'arrêter à celui précédent l'***Element*** à supprimer

Cas particuliers :

- Indice erroné (supérieur au nombre d'***Element*** ou inférieur à 1)
- Suppression du premier ***Element***

Supprimer la i^{ème} valeur d'une ***Liste***

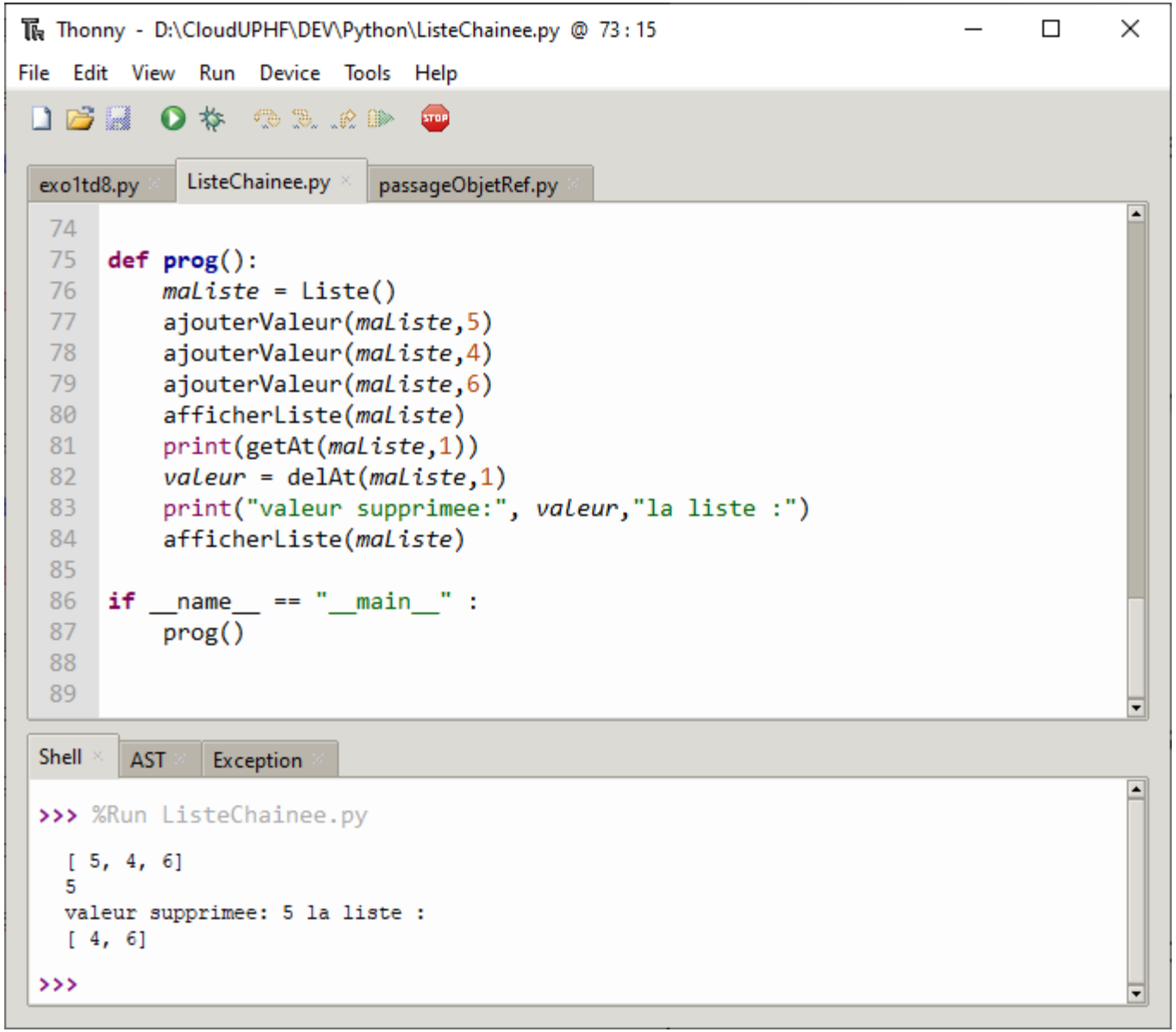
```
def delAt(lst:Liste,n:int):
    if(n>lst.nb) or n<1:
        return None
    lst.nb = lst.nb - 1
    if n==1:
        val=lst.tete.valeur
        lst.tete = lst.tete.suivant
        return val
    cpt = 1
    elt = lst.tete
    while cpt<n-1:
        elt = elt.suivant
        cpt = cpt + 1
    val = elt.suivant.valeur
    suiv = elt.suivant.suivant
    elt.suivant = suiv
    return val
```

Problème d'indice

On supprime le premier ***Element***

On parcourt tous les ***Element*** et on s'arrête avant celui à supprimer

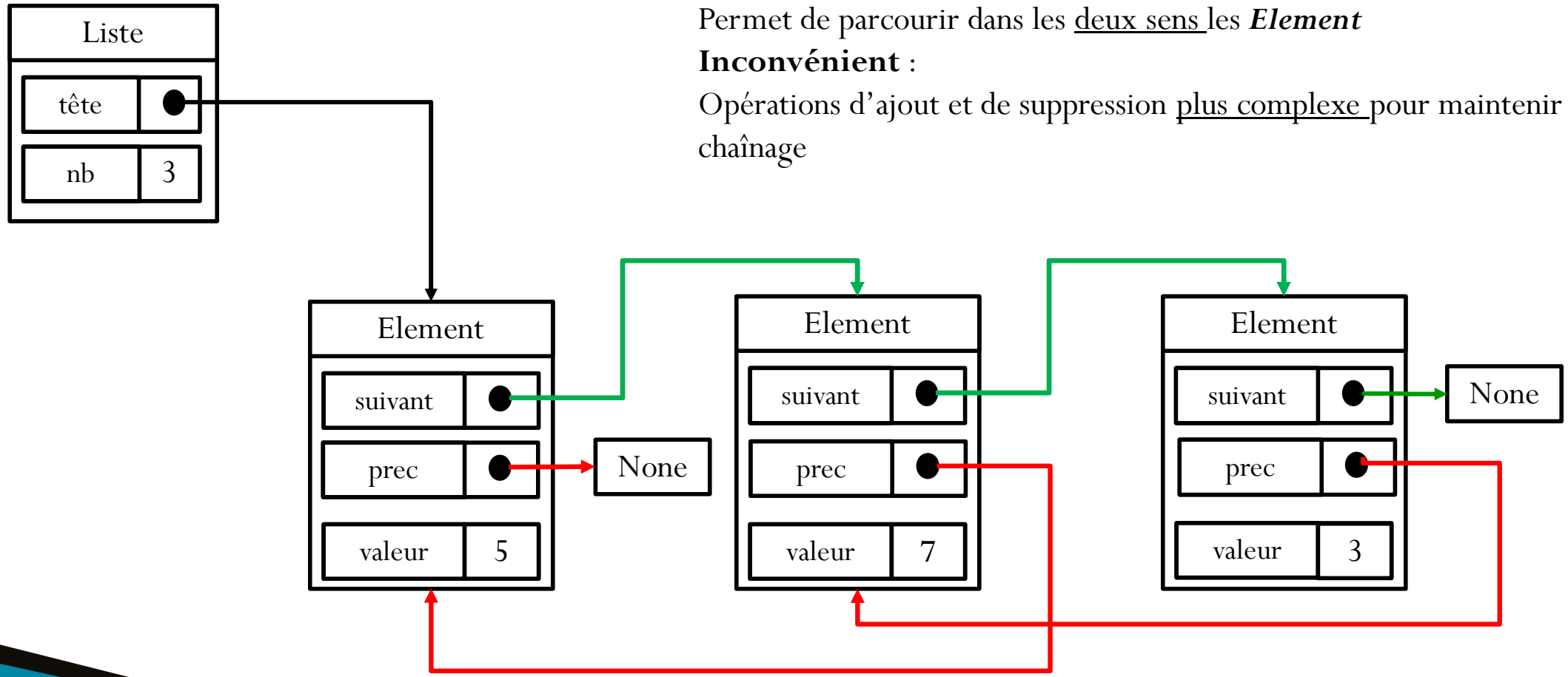
On met à jour le chaînage


Thonny - D:\CloudUPHF\DEV\Python\ListeChainee.py @ 73 : 15
File Edit View Run Device Tools Help
exo1td8.py
ListeChainee.py
passageObjetRef.py
74
75 def prog():
76 maListe = Liste()
77 ajouterValeur(maListe,5)
78 ajouterValeur(maListe,4)
79 ajouterValeur(maListe,6)
80 afficherListe(maListe)
81 print(getAt(maListe,1))
82 valeur = delAt(maListe,1)
83 print("valeur supprimee:", valeur,"la liste :")
84 afficherListe(maListe)
85
86 if __name__ == "__main__" :
87 prog()
88
89
Shell
AST
Exception
>>> %Run ListeChainee.py
[5, 4, 6]
5
valeur supprimee: 5 la liste :
[4, 6]
>>>

Variantes possibles (1)

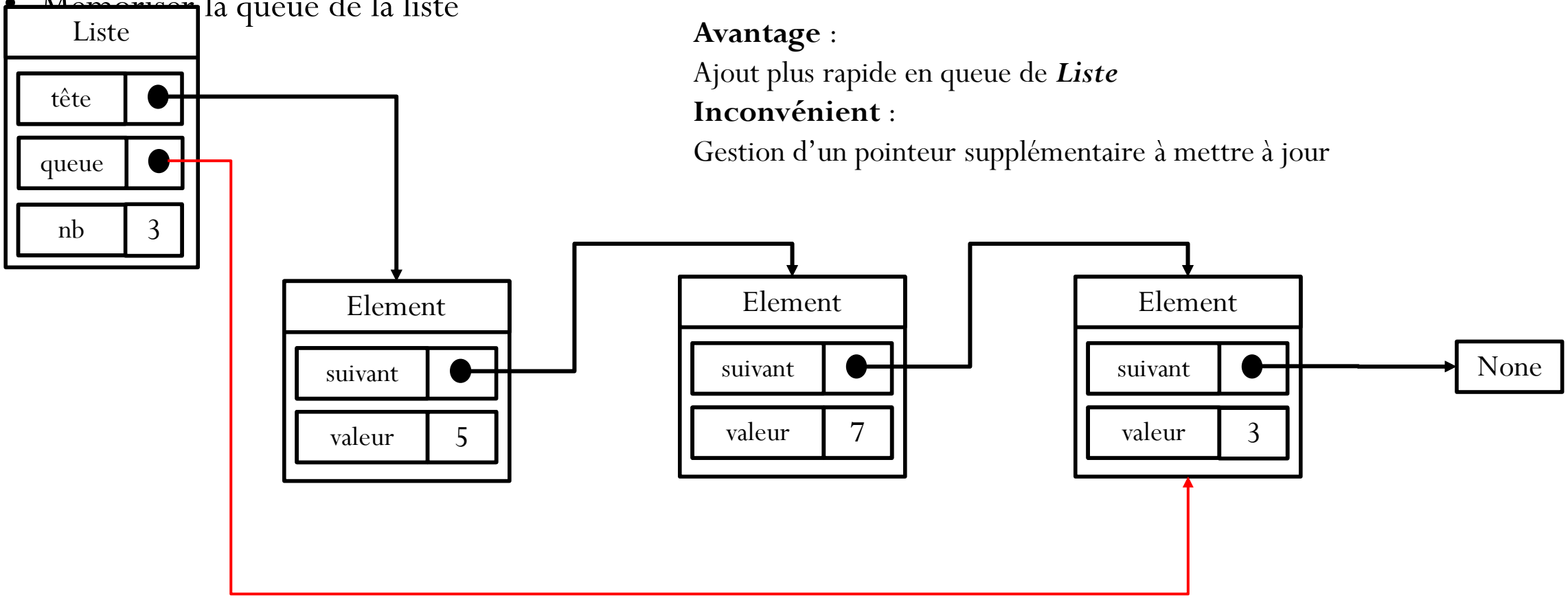
- Liste doublement chaînée:

Avantage :
Permet de parcourir dans les deux sens les ***Element***
Inconvénient :
Opérations d'ajout et de suppression plus complexe pour maintenir le chaînage



Variantes possibles (2)

- Mémoriser la queue de la liste



Avantage :
Ajout plus rapide en queue de ***Liste***
Inconvénient :
Gestion d'un pointeur supplémentaire à mettre à jour

La gestion « des erreurs »

On peut distinguer deux types « d'erreur »:

- Erreur de respect des spécifications :
  - C'est le programmeur de la fonction qui ne respectent pas correctement les spécifications
  - Temporellement on est dans le domaine de la vérification de code avant la livraison
- Cas particuliers:
  - Liés à l'usage du code qu'on a livré
  - Lié au traitement du problème (exemple résolution d'une équation du premier degré a x + b = 0, quid si a vaut 0)
  - Il ne s'agit pas d'une erreur de programmation! Mais d'un cas exceptionnel, sortant du cadre « normal »

La fonction puissance

Thonny - C:\Owncloud\DEV\Python\exempleAssertion.py @ 21 : 1

Fichier Édition Affichage Exécuter Device Outils Aide

exTp4.py exempleAssertion.py

```
1  def puissance(x:float,n:int)->float:
2      '''
3         retourne x^n
4         parametres :
5             - x un réel
6             - n un entier
7         retour:
8             retourne un réel dont la valeur est x^n
9      '''
10     if (n==0):
11         return 1
12     else :
13         return x*puissance(x,n-1)
14
15 def prog():
16     print(puissance(3,2.5))
17
18 if __name__ == "__main__":
19     prog()
```

Console

```
    return x*puissance(x,n-1)
  File "C:\Owncloud\DEV\Python\exempleAssertion.py", line 10, in puissance
    if (n==0):
RecursionError: maximum recursion depth exceeded in comparison
>>>
```

assert

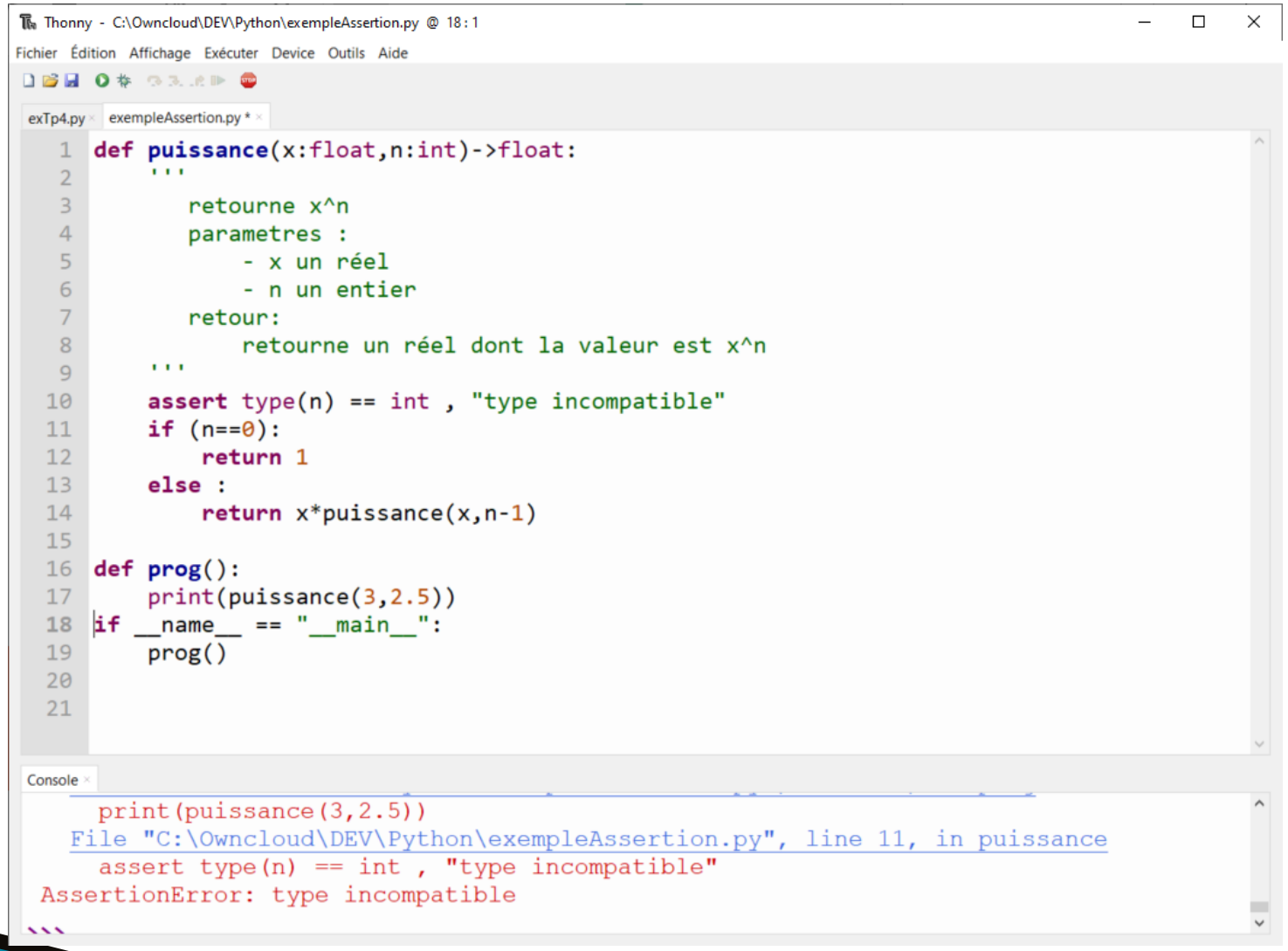
- Commande python qui s'utilise de la sorte :

  assert *expression booléenne , message (optionnel)*

  Si l'expression est vraie l'instruction ne fait rien

  Si l'expression est fausse cela conduit à l'interruption du code, affichage expliquant que l'assertion n'a pas été satisfaite (éventuellement, si un message a été spécifié, il est alors affiché)

illustration



Les assertions sont utilisées pour le debogage

```
C:\WINDOWS\system32\cmd.exe

C:\Owncloud\DEV\Python>python exempleAssertion.py
Traceback (most recent call last):
  File "exempleAssertion.py", line 19, in <module>
    prog()
  File "exempleAssertion.py", line 17, in prog
    print(puissance(3,2.5))
  File "exempleAssertion.py", line 10, in puissance
    assert type(n) == int , "type incompatible"
AssertionError: type incompatible

C:\Owncloud\DEV\Python>
```

Mode Optimal

```
C:\WINDOWS\system32\cmd.exe

C:\Owncloud\DEV\Python>python -O exempleAssertion.py
Traceback (most recent call last):
  File "exempleAssertion.py", line 19, in <module>
    prog()
  File "exempleAssertion.py", line 17, in prog
    print(puissance(3,2.5))
  File "exempleAssertion.py", line 14, in puissance
    return x*puissance(x,n-1)
  File "exempleAssertion.py", line 14, in puissance
    return x*puissance(x,n-1)
  File "exempleAssertion.py", line 14, in puissance
    return x*puissance(x,n-1)
  [Previous line repeated 994 more times]
  File "exempleAssertion.py", line 11, in puissance
    if (n==0):
RecursionError: maximum recursion depth exceeded in comparison

C:\Owncloud\DEV\Python>
```

Les exceptions en Python

Les Exceptions sont les opérations qu'un compilateur ou un interpréteur (comme python) réalise quand une erreur est détectée pendant l'exécution du programme. Dans la plupart des langages, si l'exception n'est pas gérée, le programme est alors interrompu et un message est affiché.

```
Python 3.7.2 (bundled)
>>> print(5/0)
  Traceback (most recent call last):
    File "<pyshell>", line 1, in <module>
  ZeroDivisionError: division by zero

>>>
```

Les Exceptions...

Dans de nombreux cas il est possible de prévoir certaines des erreurs qui risquent de conduire à des situations problématiques.
Dans ne nombreux langages, comme python, il est possible d'associer un mécanisme de surveillance permettant de gérer les situations anormale.

Exemple de problème

On représente un étudiant par un dictionnaire (**dict**) comportant les clefs : 'nom' (**str**), 'prenom' (**str**) et 'notes' (**list**)

[1]:
```
roger = {'nom':'Roulemapoule','prenom':'Roger','notes':[3,7,12,15]}
```

On définit une fonction (**moyenneEtu**) permettant de calculer et d'ajouter une clé 'moyenne' (**float**) à un étudiant passé en paramètre. Pour cela cette fonction fait appel à une autre fonction (**moyenne**) qui retourne la moyenne des valeurs d'une liste passée en paramètre.

[2]:
```
def moyenne(listeNotes:list)->float:
    somme : float
    somme = 0
    for note in listeNotes:
        somme = somme+note
    return somme/len(listeNotes)

def moyenneEtu(etudiant:dict)->None:
    etudiant['moyenne'] = moyenne(etudiant['notes'])
```

Calcul de la moyenne de la promotion

[3]:
```
def moyennePromo(lesEtudiants:list)->float:
    somme : float
    nbEtu : int
    somme = 0
    nbEtu = 0
    for etu in lesEtudiants:
        somme = somme + etu['moyenne']
        nbEtu = nbEtu +1
    return somme/nbEtu
```

[4]:
```
def prog():
    roger = {'nom':'Roulemapoule','prenom':'Roger','notes':[3,7,12,15]}
    moyenneEtu(roger)
    print("moyenne de Roger:",roger['moyenne'])

    albert = {'nom':'Alassoupe','prenom':'Albert','notes':[3,7,12,15]}
    bernard = {'nom':'Biengentil','prenom':'Bernard','notes':[9,9.5,12.5,11.5]}
    promo = [albert,bernard,roger]
    print('moyenne de la promo :',moyennePromo(promo))

if __name__=="__main__":
    prog()
```

Affichage:

```
moyenne de Roger: 9.25
```

```
␣
↪---------------------------------------------------------------------------

       KeyError                                  Traceback (most recent call␣
↪last)

       <ipython-input-4-3bf62ff94c5c> in <module>
        10
        11 if __name__=="__main__":
   ---> 12     prog()


       <ipython-input-4-3bf62ff94c5c> in prog()
         7     bernard = {'nom':'Biengentil','prenom':'Bernard','notes':[9,9.
↪5,12.5,11.5]}
         8     promo = [albert,bernard,roger]
   ----> 9     print('moyenne de la promo :',moyennePromo(promo))
        10
        11 if __name__=="__main__":


       <ipython-input-3-e948f15bc3e0> in moyennePromo(lesEtudiants)
         5     nbEtu = 0
         6     for etu in lesEtudiants:
   ----> 7         somme = somme + etu['moyenne']
         8         nbEtu = nbEtu +1
         9     return somme/nbEtu


       KeyError: 'moyenne'
```

Les moyennes de tous les étudiants n'ont pas été calculées…

Mécanisme des Exceptions

On peut définir en python des blocs de code « surveillé ».
Pour un bloc de code « surveillé » quand une erreur est lancée **le code de ce bloc est interrompu**, un autre bloc de code qu'on aura spécifié est exécuté.

Blocs **try** et **except**

```
def moyennePromo(lesEtudiants:list)->float:
    somme : float
    nbEtu : int
    somme = 0
    nbEtu = 0
    try:
        for etu in lesEtudiants:
            somme = somme + etu['moyenne']
            nbEtu = nbEtu +1
        return somme/nbEtu

    except Exception as error:
        print("error-->",error)
        print("calcul des moyennes manquantes...")
        for etu in lesEtudiants:
            if not 'moyenne' in etu:
                print("traitement de :",etu['nom'])
                moyenneEtu(etu)

        return moyennePromo(lesEtudiants)
```

Bloc susceptible de lever une erreur (Exception)
Bloc « try: »

Bloc exécuté en cas d'exception levée
Bloc « except Exception: » (le « as error » est optionnel)

```
moyenne de Roger: 9.25
error--> 'moyenne'
calcul des moyennes manquantes...
traitement de : Alassoupe
traitement de : Biengentil
moyenne de la promo : 9.708333333333334
```

Détecter un problème et lever une exception

```
def moyennePromo(lesEtudiants:list)->float:
    somme : float
    nbEtu : int
    somme = 0
    nbEtu = 0
    try:
        for etu in lesEtudiants:
            if not 'moyenne' in etu:
                raise Exception("pas de moyenne!!!")
            somme = somme + etu['moyenne']
            nbEtu = nbEtu +1
        return somme/nbEtu

    except Exception as error:
        print("error-->",error)
        print("calcul des moyennes manquantes...")
        for etu in lesEtudiants:
            if not 'moyenne' in etu:
                print("traitement de :",etu['nom'])
                moyenneEtu(etu)
        return moyennePromo(lesEtudiants)
```

Détection du problème (remarquez au passage comment on teste la présence d'une valeur dans une collection)

L'instruction **raise Exception(« message »)** permet de lever l'Exception

```
moyenne de Roger: 9.25
error--> pas de moyenne!!!
calcul des moyennes manquantes...
traitement de : Alassoupe
traitement de : Biengentil
moyenne de la promo : 9.708333333333334
```

Appeler une fonction susceptible de lever une Exception

Thonny - C:\Owncloud\DEV\Python\exempleAssertion.py @ 35 : 1

Fichier Édition Affichage Exécuter Device Outils Aide

exTp4.py | exempleAssertion.py

```
21
22 def solveEq1(a,b):
23     if (a==0):
24         raise Exception("coefficient degre 1 nul")
25     else :
26         return -b/a
27
28 def prog2():
29     va = float(input("a?"))
30     vb = float(input("b?"))
31     try:
32         x = solveEq1(va,vb)
33         print("la solution est : ", x)
34
35     except Exception as e:
36         print("l equation n'admet pas de solution")
37
38
39 if __name__ == "__main__":
40     prog2()
41
42
```

Console

```
a?0
b?3
l equation n'admet pas de solution
>>>
```

Introduction aux fichiers en Python

CHAPITRE 5

Qu'est-ce qu'un fichier?

Un fichier est un ensemble d'octets sauvegardé sur un support (la disquette au temps de la préhistoire, la clée USB, le CD-ROM, DVD-ROM, Blue Ray, disque dur HDD/SDD…).

Tout comme lors de l'exécution d'un programme (donc durant cette exécution) des informations sont stockées en mémoire vive, il est possible de stocker des informations sur un « disque » dans un contenant qui est un fichier.

Avantages :

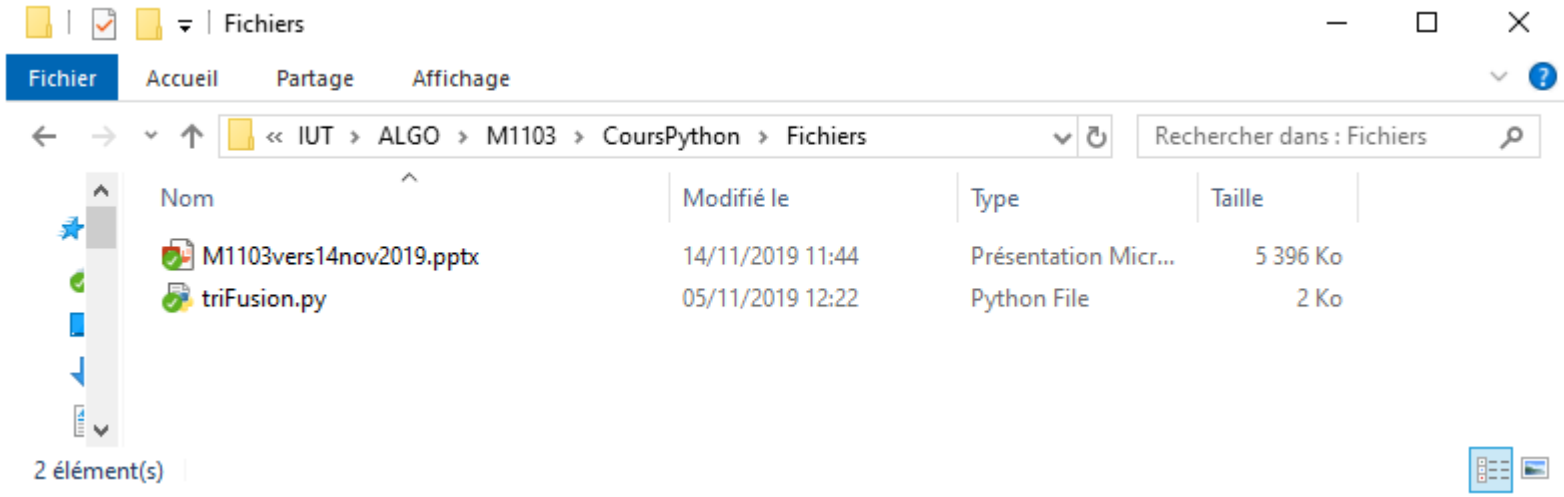
- Persistance des données : même après l'arrêt du programme des données peuvent être stockées et donc récupérées par la suite.
- Echange d'information entre plusieurs programmes
- Echange d'information entre plusieurs utilisateurs
- Accès distant à des informations

Format de fichier

Comme pour les variables il y a plusieurs types de fichiers.

On distingue 2 grandes familles de fichiers :

- Les fichiers « texte »
- Les fichiers « binaire »



Fichier « texte »

D:\CloudUPHF\IUT\ALGO\M1103\CoursPython\Fichiers\triFusion.py - Notepad++

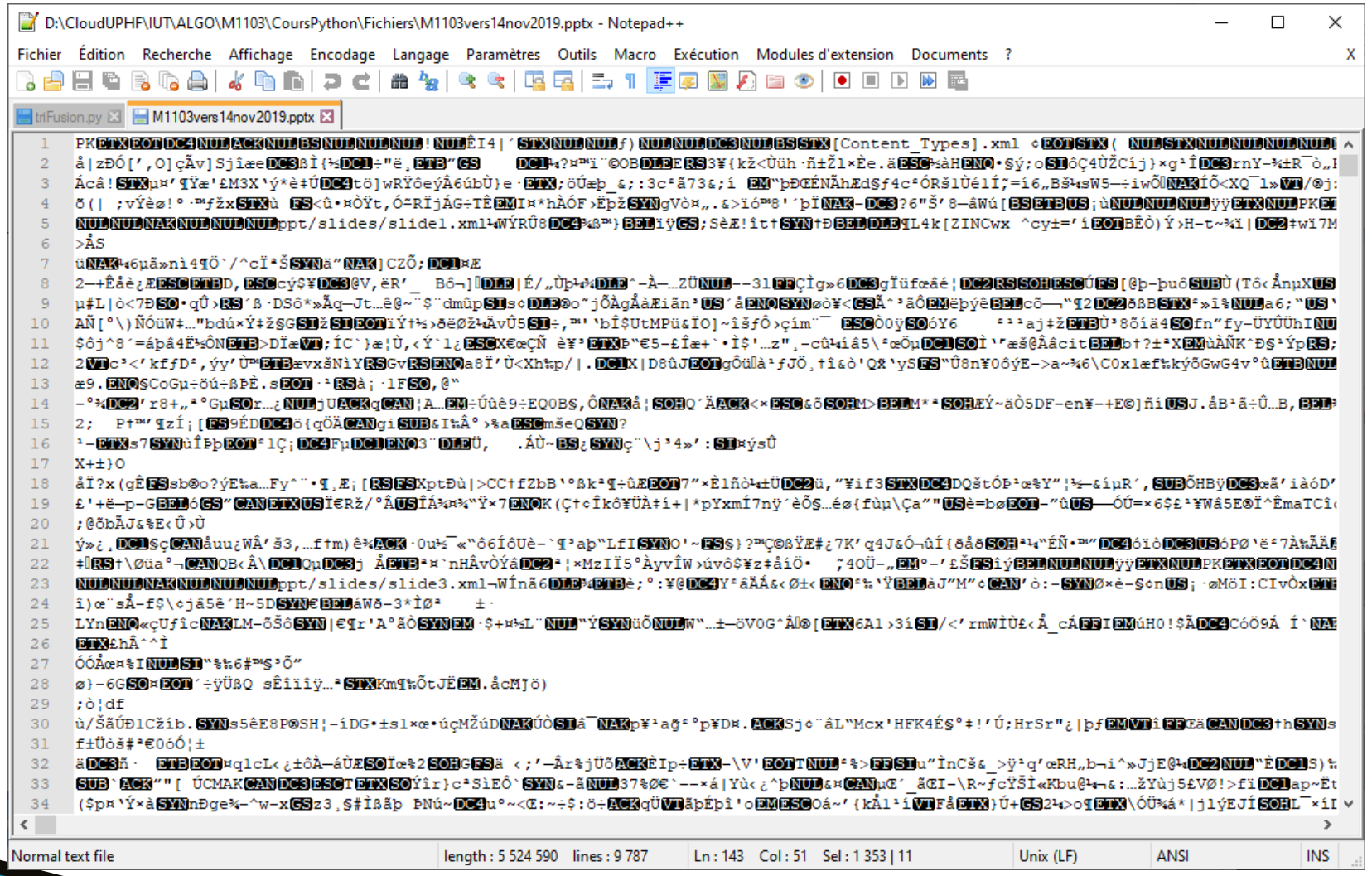
Fichier Édition Recherche Affichage Encodage Langage Paramètres Outils Macro Exécution Modules d'extension Documents ? X

triFusion.py

```
1
2  def remplacer(liste:list, deb:int, nouv:list):
3      i = 0
4      while i<len(nouv):
5          liste[deb+i] = nouv[i]
6          i = i+1
7
8
9  def fusionner(liste : list,deb:int, mil: int, fin: int):
10     j:int
11     k:int
12     j,k = deb,mil+1
13     tmp = []
14     while j<=mil and k<=fin:
15         if liste[j]<liste[k]:
16             tmp.append(liste[j])
17             j = j+1
18         else:
19             tmp.append(liste[k])
20             k = k+1
21     while j<=mil:
22         tmp.append(liste[j])
23         j = j+1
24     while k<=fin:
25         tmp.append(liste[k])
26         k = k+1
27
28     remplacer(liste,deb,tmp)
29
30
31 def trierPartition(liste:list, deb:int, fin:int):
32     if(fin-deb>0):
33         if(fin-deb==1) and liste[deb]>liste[fin]:
```

Python file | length : 1 138 lines : 47 | Ln : 1 Col : 1 Sel : 0 | 0 | Windows (CR LF) | UTF-8 | INS

Fichiers « binaire »



Avantages/inconvénients fichiers « texte »

L'information est directement accessible : +/- comme la saisie au clavier

Facilement modifiable par une application tierce → problème de corruption de données

Attention au codage des caractères: différentes normes existent (ASCII, UTF-8, ISO/CEI 8859-15…) ceci peut poser des problèmes pour « décrypter un fichier texte ».

Concernant le codage des caractères voir l'article dédié de WIKIPEDIA :
https://fr.wikipedia.org/wiki/Codage_des_caractères

Avantages/inconvénients fichiers « binaire »

- L'information n'est pas directement accessible, seul celui qui a développé l'application qui a généré le fichier sait comment est structuré/codé le fichier
- Récupération (et écriture) des données directement : pas besoin de convertir une chaîne de caractères vers le type de données.
- Corruption de données moins préjudiciable

Ecriture dans un fichier texte en Python

3 étapes :

1. création ou ouverture du fichier,
2. Écriture de données,
3. Fermeture du fichier.

Lors de l'ouverture, le fichier dans lequel seront écrites les informations est créé, s'il n'existe pas, ou nettoyé s'il existe déjà. La fermeture permet à d'autres programmes de lire ce que vous avez placé dans ce fichier. Sans cette dernière étape, il sera impossible d'y accéder à nouveau pour le lire ou y écrire à nouveau. A l'intérieur d'un programme informatique, écrire dans un fichier suit toujours le même schéma :

```
f = open ("nom-fichier", "w")     # ouverture

f.write ( s )     # écriture de la chaîne de caractères  s
f.write ( s2 )    # écriture de la chaîne de caractères  s2
...

f.close ()  # fermeture
```

Avec le bloc **with**, a la sortie du bloc, le fichier est automatiquement fermé

```
with open ("nom-fichier", "w") as f:     # ouverture

    f.write ( s )     # écriture de la chaîne de caractères  s
    f.write ( s2 )    # écriture de la chaîne de caractères  s2
    ...
```

Exemple sauvegarde d'une matrice

Certains caractères sont fort utiles lors de l'écriture de fichiers texte afin d'organiser les données. Le symbole ; est très utilisé comme séparateur de colonnes pour une matrice, on utilise également le passage à la ligne ou la tabulation. Comme ce ne sont pas des caractères « visibles », ils ont des codes :

- \n : passage à la ligne
- \t : tabulation, indique un passage à la colonne suivante

```
mat =  ... # matrice de type liste de listes
with open ("mat.txt", "w") as f:
    for i in range (0,len (mat)) :
        for j in range (0, len (mat [i])) :
            f.write ( str (mat [i][j]) + "\t")
        f.write ("\n")
```

La fonction *open()*

La fonction open accepte **2 paramètres**, le premier est le **nom du fichier**, le second définit le **mode d'ouverture** : "**w**" pour écrire (****w****rite), « **a** » pour écrire et ajouter (****a****ppend), « **r** » pour lire (****r****ead). Ceci signifie que la fonction open sert à ouvrir un fichier quelque soit l'utilisation qu'on en fait.

```
with open ("essai.txt", "w") as f:
    f.write (" premiere fois ")


with f = open ("essai.txt", "w") as f:
    f.write (" seconde fois ")
```

écrasement

```
with open ("essai.txt", "w") as f:
    f.write (" premiere fois ")


with f = open ("essai.txt", "a") as f:  ###
    f.write (" seconde fois ")
```

ajout

Lire dans un fichier « texte »

La lecture d'un fichier permet de retrouver les informations stockées grâce à une étape préalable d'écriture. Elle se déroule selon le même principe, à savoir :

1. ouverture du fichier en mode lecture,
2. lecture,
3. fermeture.

```
nom  ; prénom ; livre
Hugo  ; Victor  ; Les misérables
Kessel ; Joseph  ; Le lion
Woolf ; Virginia  ; Mrs Dalloway
Calvino ; Italo  ; Le baron perché
```

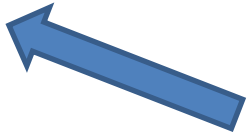
Une différence apparaît cependant lors de la lecture d'un fichier : celle-ci s'effectue ligne par ligne alors que l'écriture ne suit pas forcément un découpage en ligne.

```
mat = []                                # création d'une liste vide,
with open ("essai.txt", "r") as f:      # ouverture du fichier en mode lecture
    for li in f :                       # pour toutes les lignes du fichier
        s = li.strip ("\n\r")           # on enlève les caractères de fin de ligne
        l = s.split (";")               # on découpe en colonnes
        mat.append (l)                  # on ajoute la ligne à la matrice
```

Lire une ligne dans un fichier texte

```
f = open (filename, "r")
uneLigne = f.readline()
```

uneLigne est un objet de classe **str**



readline() est une méthode, elle est appelée à partir d'un objet de classe **TextIOWrapper** (classe permettant de manipuler un fichier texte en lecture)

Ecriture dans un fichier « binaire »

Il existe un moyen de sauvegarder dans un fichier des objets plus complexes à l'aide du module **pickle**. Celui-ci permet de stocker dans un fichier le contenu d'un objet. Le principe pour l'écriture est le suivant :

Utilisation de la méthode ***dump*** de *pickle*

```
import pickle

dico = {'a': [1, 2.0, 3, "e"], 'b': ('string', 2), 'c': None}
lis  = [1, 2, 3]

with open ('data.bin', 'wb') as fb:
    pickle.dump(dico, fb)
    pickle.dump(lis, fb)
```

Lecture d'un fichier « binaire »

Utilisation de la méthode *load* de *pickle*

```
with open('data.bin', 'rb') as fb:
    dico = pickle.load(fb)
    lis  = pickle.load(fb)
```

Université Polytechnique HAUTS-DE-FRANCE

Les fichiers CSV

CHAPITRE 6

Table

Une table représente une collection d'éléments (données tabulées). Chaque ligne représente un élément de la collection. A chaque colonne correspond un attribut. La table ci-dessous représente un ensemble d'étudiants. Chaque étudiant a:

- Un nom
- Un prénom
- Une promotion
- Une note correspondant à la moyenne générale

Nom	Prenom	Promotion	Moyenne
Durant	Jean	1	12.5
Dupont	Pierre	1	9.5
Lenoir	Roger	2	10
Lerouge	Arthur	2	11
Legris	Guillaume	1	12

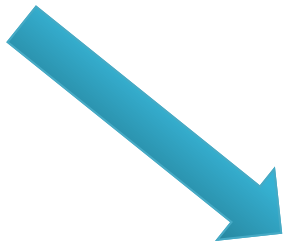
Fichier et tables (données tabulées)

Pour qu'un programme puisse échanger des données tabulées avec un autre programme (ou lui-même) on utilise généralement des fichiers dans un format de stockage standard. Le standard le plus courant est CSV (Comma-Separed Values, « Données Séparées par des Virgules »). C'est un format simple puisque:

- Les fichier CSV sont des fichiers textes;
- Chaque ligne du fichier correspond à une ligne de la table;
- Chaque ligne est séparée en champs au moyen du caractère « , »;
- Toutes les lignes du fichier ont le même nombre de champs;
- La première ligne du fichier peut représenter les noms d'attributs.

Table en fichier CSV

Nom	Prenom	Promotion	Moyenne
Durant	Jean	1	12.5
Dupont	Pierre	1	9.5
Lenoir	Roger	2	10
Lerouge	Arthur	2	11
Legris	Guillaume	1	12



etudiant.csv

```
Nom,Prenom,Promotion,Moyenne
Durant,Jean,1,12.5
Dupont,Pierre,1,9.5
Lenoir,Roger,2,10
Lerouge,Arthur,2,11
Legris,Guillaume,1,12
```

Récupérer les données d'un fichier CSV (1)

Lire le fichier et reconstruire les données lignes par lignes….

```
with open("etudiants.csv","r") as fichier:
    lesLignes = fichier.readlines()

entete = lesLignes[0].strip("\n\r").split(",")
print("liste des attributs:",entete)
print("-------------------------")
donnees=[]
for numLigne in range(1,len(lesLignes)):
    donnees.append(lesLignes[numLigne].strip("\n\r").split(","))
print("liste des etudiants: ",donnees)
print("-------------------------")
table=[]
for etudiant in donnees:
    dico ={}
    for i in range(len(entete)):
        dico[entete[i]] = etudiant[i]
    table.append(dico)
print("la liste de dictionnaires : ",table)
print("-------------------------")
```

```
>>> %Run tables.py
liste des attributs: ['Nom', 'Prenom', 'Promotion', 'Moyenne']
-------------------------
liste des etudiants:  [['Durant', 'Jean', '1', '12.5'],
['Dupont', 'Pierre', '1', '9.5'], ['Lenoir', 'Roger', '2',
'10'], ['Lerouge', 'Arthur', '2', '11'], ['Legris',
'Guillaume', '1', '12']]
-------------------------
la liste de dictionnaires :  [{'Nom': 'Durant', 'Prenom':
'Jean', 'Promotion': '1', 'Moyenne': '12.5'}, {'Nom':
'Dupont', 'Prenom': 'Pierre', 'Promotion': '1', 'Moyenne':
'9.5'}, {'Nom': 'Lenoir', 'Prenom': 'Roger', 'Promotion': '2',
'Moyenne': '10'}, {'Nom': 'Lerouge', 'Prenom': 'Arthur',
'Promotion': '2', 'Moyenne': '11'}, {'Nom': 'Legris',
'Prenom': 'Guillaume', 'Promotion': '1', 'Moyenne': '12'}]
-------------------------
```

Récupérer les données d'un fichier CSV (2)

Utiliser le module csv

```
import csv
with open("etudiants.csv","r") as fichier:
    mesDonnees = list(csv.DictReader(fichier))
print("dictionnaire ordonné récupéré :", mesDonnees)
print("------------------------------")
print(mesDonnees[1]["Nom"])
```

```
dictionnaire ordonné récupéré :
[OrderedDict([('Nom', 'Durant'), ('Prenom', 'Jean'),
('Promotion', '1'), ('Moyenne', '12.5')]),
OrderedDict([('Nom', 'Dupont'), ('Prenom',
'Pierre'), ('Promotion', '1'), ('Moyenne', '9.5')]),
OrderedDict([('Nom', 'Lenoir'), ('Prenom', 'Roger'),
('Promotion', '2'), ('Moyenne', '10')]),
OrderedDict([('Nom', 'Lerouge'), ('Prenom',
'Arthur'), ('Promotion', '2'), ('Moyenne', '11')]),
OrderedDict([('Nom', 'Legris'), ('Prenom',
'Guillaume'), ('Promotion', '1'), ('Moyenne',
'12')])]
------------------------------
Dupont
```

Dictionnaire Ordonné (OrderedDict)

OrderedDict : Une classe Python qui a peu de différence avec les Dict (depuis Python 3.6)

```
from collections import OrderedDict

un = OrderedDict()
un["arabe"] = 1
un["latin"] = 'I'

unbis = OrderedDict()
unbis["latin"] = 'I'
unbis["arabe"] = 1


uno = {}
uno["arabe"] = 1
uno["latin"] = 'I'

unobis = {}
unobis["latin"] = 'I'
unobis["arabe"] = 1

print(un, unbis)
print(uno,unobis)

print("uno == unobis?", uno==unobis)
print("un == unbis?", un==unbis)
```

```
OrderedDict([('arabe', 1), ('latin', 'I')])
OrderedDict([('latin', 'I'), ('arabe', 1)])
{'arabe': 1, 'latin': 'I'} {'latin': 'I', 'arabe': 1}
uno == unobis? True
un == unbis? False
```

Gestion des types

Dans notre exemple les données de la table ont des types différents, or quand on les récupère via le fichier tout est chaîne de caractères (str)

str	str	int	float

Nom	Prenom	Promotion	Moyenne
Durant	Jean	1	12.5
Dupont	Pierre	1	9.5
Lenoir	Roger	2	10
Lerouge	Arthur	2	11
Legris	Guillaume	1	12

```
dictionnaire ordonné récupéré : [OrderedDict([('Nom',
'Durant'), ('Prenom', 'Jean'), ('Promotion', '1'),
('Moyenne', '12.5')]), OrderedDict([('Nom', 'Dupont'),
('Prenom', 'Pierre'), ('Promotion', '1'), ('Moyenne',
'9.5')]), OrderedDict([('Nom', 'Lenoir'), ('Prenom',
'Roger'), ('Promotion', '2'), ('Moyenne', '10')]),
OrderedDict([('Nom', 'Lerouge'), ('Prenom', 'Arthur'),
('Promotion', '2'), ('Moyenne', '11')]),
OrderedDict([('Nom', 'Legris'), ('Prenom', 'Guillaume'),
('Promotion', '1'), ('Moyenne', '12')])]
```

Conversion

```
def typage(unElement: OrderedDict) -> OrderedDict:
    nouv = OrderedDict()
    nouv["Nom"] = unElement["Nom"]
    nouv["Prenom"] = unElement["Prenom"]
    nouv["Promotion"] = int(unElement["Promotion"])
    nouv["Moyenne"] = float(unElement["Moyenne"])
    return nouv

def convertir(laTable:list, laFonction) -> list:
    donneesValides = []
    for etu in laTable:
        donneesValides.append(laFonction(etu))
    return donneesValides

print(mesDonnees)
mesDonneesValides = convertir(mesDonnees,typage)
print(mesDonneesValides)
```

```
dictionnaire ordonné récupéré : [OrderedDict([('Nom', 'Durant'),
('Prenom', 'Jean'), ('Promotion', '1'), ('Moyenne', '12.5')]),
OrderedDict([('Nom', 'Dupont'), ('Prenom', 'Pierre'),
('Promotion', '1'), ('Moyenne', '9.5')]), OrderedDict([('Nom',
'Lenoir'), ('Prenom', 'Roger'), ('Promotion', '2'), ('Moyenne',
'10')]), OrderedDict([('Nom', 'Lerouge'), ('Prenom', 'Arthur'),
('Promotion', '2'), ('Moyenne', '11')]), OrderedDict([('Nom',
'Legris'), ('Prenom', 'Guillaume'), ('Promotion', '1'),
('Moyenne', '12')])]

[OrderedDict([('Nom', 'Durant'), ('Prenom', 'Jean'),
('Promotion', 1), ('Moyenne', 12.5)]), OrderedDict([('Nom',
'Dupont'), ('Prenom', 'Pierre'), ('Promotion', 1), ('Moyenne',
9.5)]), OrderedDict([('Nom', 'Lenoir'), ('Prenom', 'Roger'),
('Promotion', 2), ('Moyenne', 10.0)]), OrderedDict([('Nom',
'Lerouge'), ('Prenom', 'Arthur'), ('Promotion', 2), ('Moyenne',
11.0)]), OrderedDict([('Nom', 'Legris'), ('Prenom',
'Guillaume'), ('Promotion', 1), ('Moyenne', 12.0)])]
```

Programme complet

```python
from collections import OrderedDict
import csv

def typage(unElement: OrderedDict) -> OrderedDict:
    nouv = OrderedDict()
    nouv["Nom"] = unElement["Nom"]
    nouv["Prenom"] = unElement["Prenom"]
    nouv["Promotion"] = int(unElement["Promotion"])
    nouv["Moyenne"] = float(unElement["Moyenne"])
    return nouv

def convertir(laTable:list, laFonction) -> list:
    donneesValides = []
    for etu in laTable:
        donneesValides.append(laFonction(etu))
    return donneesValides

def importerDonnees(nomFichier:str)->OrderedDict:
    with open(nomFichier,"r") as fichier:
        mesDonnees = list(csv.DictReader(fichier,delimiter=","))
    return convertir(mesDonnees,typage)

if __name__ == '__main__' :
    print("Donnees importées:")
    data  = importerDonnees("etudiants.csv")
    for etu in data:
        print("|",end=' ')
        for cle in etu:
            print(cle, ":", etu[cle],end="\t|")
        print()
```

Donnees importées:

| Nom : Durant	|Prenom : Jean	|Promotion : 1	|Moyenne : 12.5 |
| Nom : Dupont	|Prenom : Pierre	|Promotion : 1	|Moyenne : 9.5 |
| Nom : Lenoir	|Prenom : Roger	|Promotion : 2	|Moyenne : 10.0 |
| Nom : Lerouge	|Prenom : Arthur	|Promotion : 2	|Moyenne : 11.0 |
| Nom : Legris	|Prenom : Guillaume	|Promotion : 1	|Moyenne : 12.0 |

Exporter vers un fichier CSV

```
def exporter(nomFichier:str,data:list):
    with open(nomFichier,"w") as fichier:
        w = csv.DictWriter(fichier,data[0].keys())
        w.writeheader()
        w.writerows(data)
```

On récupère automatiquement les clés

On écrit la première ligne des entêtes

« visualiser » les variables et objets: introduction à l'outil en ligne python tutor

CHAPITRE 6

Où sont mes variables?

Nous avions vu dans un chapitre précédent que les « variables locales » sont stockées dans le contexte de la fonction à laquelle elles appartiennent.

Ces variables sont des variables dites statiques et ne sont accessibles que dans le contexte dans lequel elles sont.

Plus tard nous avons vu les **list** qui semblent être modifiées en dehors de leur contexte.

```
def ajouter(unEntier, uneValeur):
    unEntier = unEntier+uneValeur

def prog():
    x = 12
    print(x)
    ajouter(x,2)
    print(x)

if __name__ == "__main__":
    prog()
```

Dans prog, x n'est pas modifié après l'exécution de la fonction ajouter

```
12
12
```

```
def ajouter(uneListe, uneValeur):
    uneListe.append(uneValeur)

def prog():
    x = [12]
    print(x)
    ajouter(x,2)
    print(x)

if __name__ == "__main__":
    prog()
```

Dans prog, x est modifié après l'exécution de la fonction ajouter

```
[12]
[12, 2]
```

Le tas (heap) / la pile (stack)

Université Polytechnique HAUTS-DE-FRANCE

Le tas:

La mémoire pour stocker un objet de class list est *allouée dynamiquement*, cette zone mémoire se situe dans la zone appelée « *tas* » (*heap*). Le tas est une zone mémoire accessible à tous les contextes. Tous les objets « mutables » sont stockés dans le tas.
Ceci sera vu normalement plus en détail en système au Semestre 2.
La taille du tas est limité à la capacité maximum de la mémoire de la machine (mémoire virtuelle comprise). L'allocation (réservation) et la désallocation (libération) de l'espace mémoire des variables dans le tas est géré par le programme.

La pile:

La pile est une zone mémoire qui comme son nom l'indique désalloue en premier le dernier élément alloué. Les contextes des fonctions, que nous avons vus précédement, sont gérés par la pile
Dans la pile les variables ne sont accessibles qu'à l'intérieur de leur contexte.
L'allocation et la désallocation de la mémoire est gérée automatiquement.
Dans les langages compilés : l'accès mémoire à la pile est plus rapide que dans le tas. La taille de la pile est limité (dépend du Système d'Exploitation).

Simulation:

Pour « visualiser » la mémoire lors de l'exécution (Pile et Tas) , on peut utiliser pythontutor : qui est un simulateur en ligne:
http://www.pythontutor.com/
Vous devez être capable de simuler par vous-même l'évolution de la pile et du tas (schématiquement)

Pile et contextes

La pile est organisée en contextes :

- Le contexte principal,
- Un contexte supplémentaire pour chaque appel de fonction en cours d'exécution

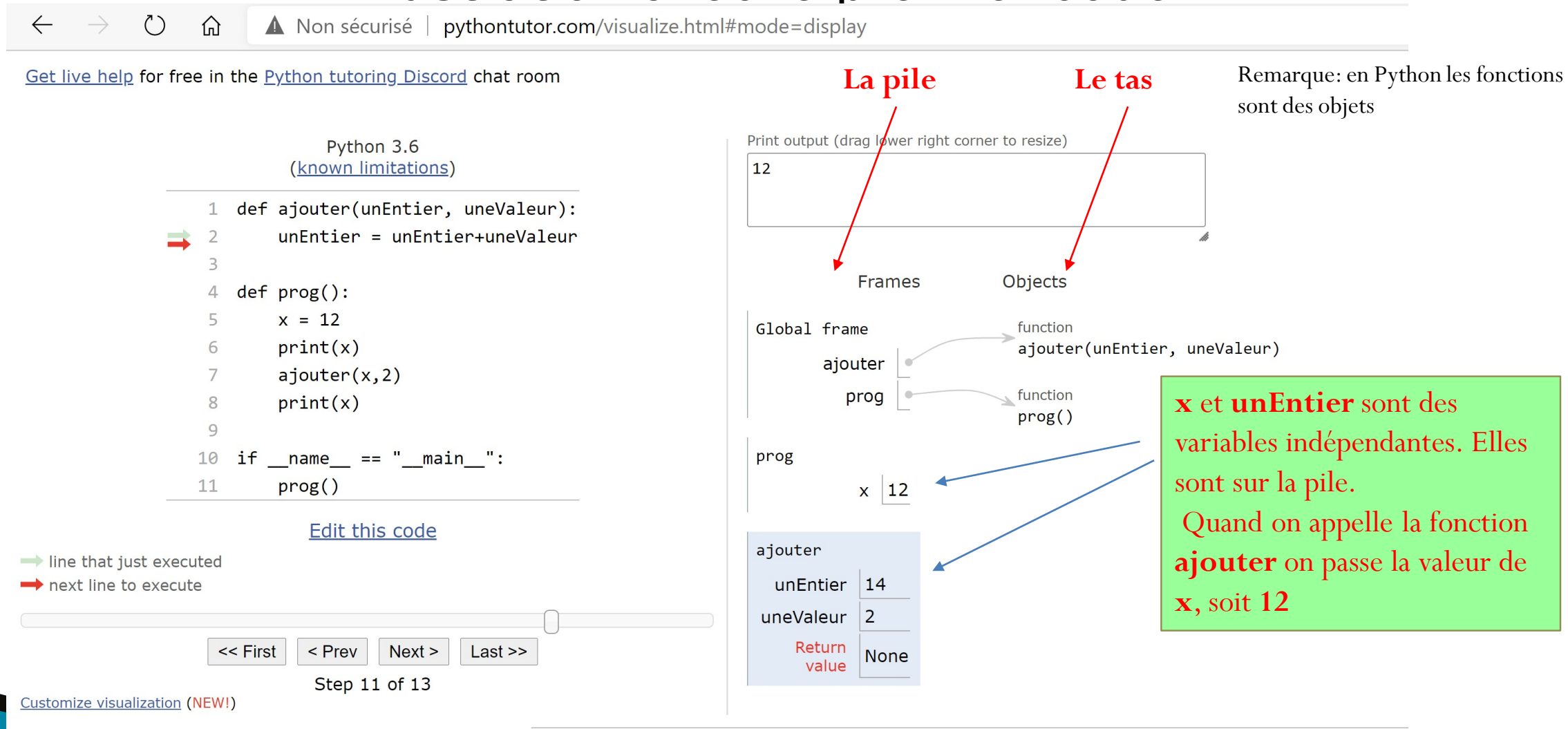
Chaque contexte :

- Contient les paramètres,
- Les variables utilisées par la fonction,
- Les informations sur la position courante dans le code (généralement pas représentée dans les simulateurs), qui permet à la fin de l'exécution de la fonction de continuer le flot d'exécution.

L'exécution de l'instruction return ou de la dernière instruction de la fonction:

- Le contexte en cours est supprimé
- Le contexte précédent est restauré

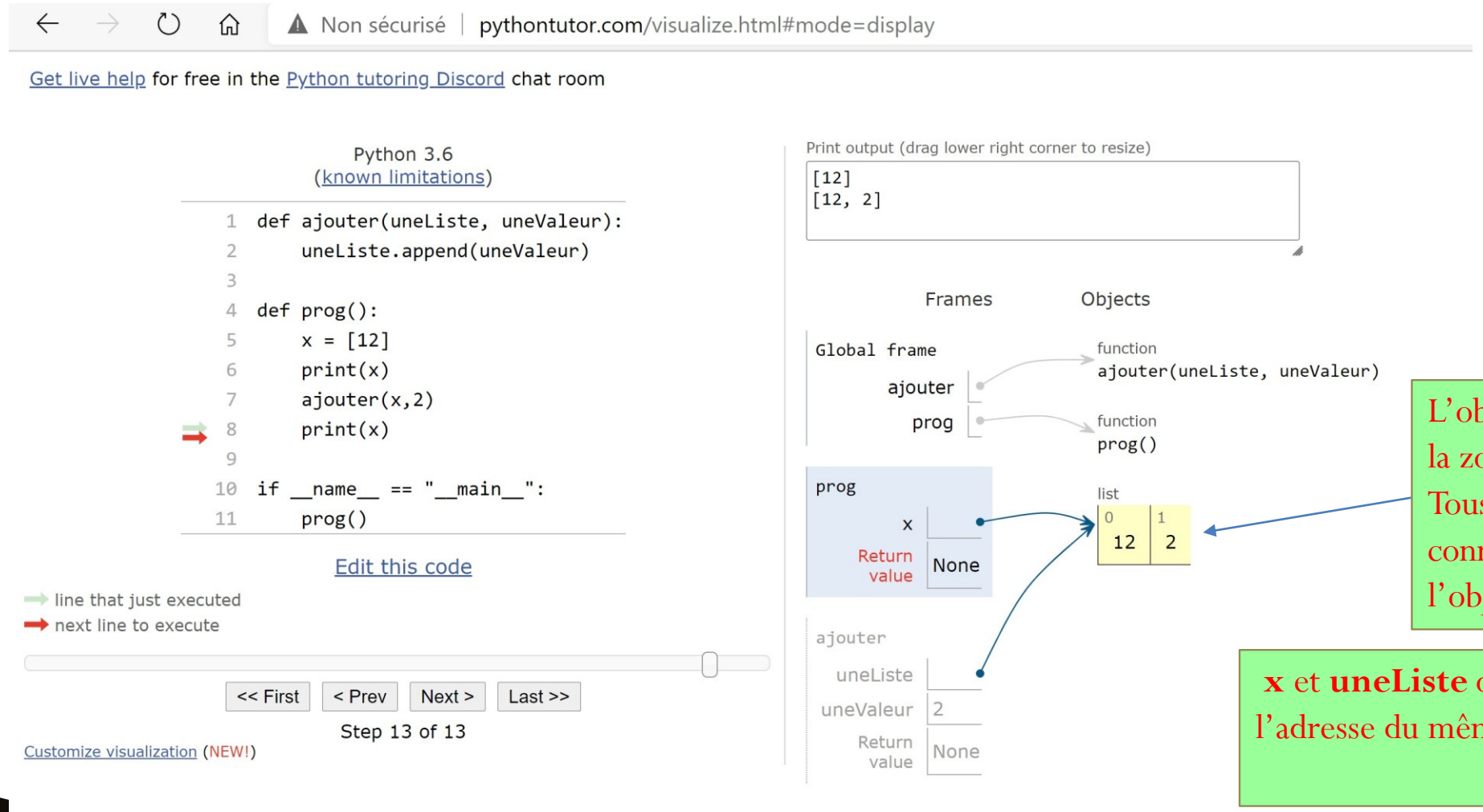
Illustration avec le premier code



Remarque: en Python les fonctions sont des objets

x et **unEntier** sont des variables indépendantes. Elles sont sur la pile.
Quand on appelle la fonction **ajouter** on passe la valeur de **x**, soit **12**

Illustration avec le deuxième code



L'objet **list** est stocké dans la zone du tas.
Tous les contextes qui connaissent l'adresse de l'objet y ont accès.

x et **uneListe** ont pour valeur l'adresse du même objet **list**

LES LISTES (LIST) DE COMPRÉHENSION EN PYTHON

Créer une liste de 100 éléments de 1 à 100

```
def creerListe(debut:int,fin:int)->list:
    res = []
    for v in range(debut,fin+1):
        res.append(v)
    return res


if __name__ == "__main__":
    maListe = creerListe(1,100)
```

```
>>> %Run comprehension.py
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51,
52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,
68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83,
84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99,
100]
>>>
```

Lourdeur d ’écriture!

Compréhension de liste : définition

- Une compréhension de liste consiste à placer entre crochets une expression suivie par une clause **for** puis par zéro ou plus clauses **for** ou **if**.
- Le résultat est une nouvelle liste résultat de l'évaluation de l'expression dans le contexte des clauses **for** et **if** qui la suivent.

Application à notre problème

```
if __name__ == "__main__":
    #maListe = creerListe(1,100)
    maListe = [x for x in range(1,101)]
    print(maListe)
```

```
>>> %Run comprehension.py
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57,
58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75,
76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93,
94, 95, 96, 97, 98, 99, 100]
>>>
```

Ne filtrer que les nombres pairs

```
if __name__ == "__main__":
    #maListe = creerListe(1,100)
    maListe = [x for x in range(1,101)]
    lesPairs = [x for x in maListe if x%2==0]
    print(maListe)
    print(lesPairs)
```

```
>>> %Run comprehension.py
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57,
58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75,
76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93,
94, 95, 96, 97, 98, 99, 100]
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38,
40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74,
76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 100]
>>>
```

Appliquer un traitement à tous les éléments d'une liste

```
if __name__ == "__main__":
    #maListe = creerListe(1,100)
    maListe = [x for x in range(1,101)]
    lesPairs = [x for x in maListe if x%2==0]
    print(maListe)
    print(lesPairs)
    lesPairsCarre = [x**2 for x in lesPairs]
    print(lesPairsCarre)
```

```
>>> %Run comprehension.py
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57,
58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75,
76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93,
94, 95, 96, 97, 98, 99, 100]
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38,
40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74,
76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 100]
[4, 16, 36, 64, 100, 144, 196, 256, 324, 400, 484, 576, 676, 784, 900,
1024, 1156, 1296, 1444, 1600, 1764, 1936, 2116, 2304, 2500, 2704,
2916, 3136, 3364, 3600, 3844, 4096, 4356, 4624, 4900, 5184, 5476,
5776, 6084, 6400, 6724, 7056, 7396, 7744, 8100, 8464, 8836, 9216,
9604, 10000]
```

Cascade de clauses

Les cubes des nombres pairs de 1 à 10 qui sont divisibles par 3 :

```
if __name__ == "__main__":
    maListe = [x**3 for x in [y for y in range(1,11) if y%2==0] if (x**3)%3==0]
    print(maListe)
```

Liste des nombres pairs compris entre 1 et 10

Table de vérité du ET logique (and)

```
if __name__ == "__main__":
    maListe = [(x,y, x and y ) for x in [True,False] for y in [True,False]]
    print(maListe)
```

```
>>> %Run comprehension.py
[(True, True, True), (True, False, False), (False, True, False), (False, False, False)]
>>>
```

« multiplication » list

- maListe = [1,2]*3
- >>>[1,2,1,2,1,2]

Matrice et compréhension de list

- On veut créer une matrice de nbL lignes nbC colonnes dont les valeurs vont de 1 à nbL*nbC
-
```
def creerMatrice(nbL,nbC):
  return [[x for x in range(nbC*i+1,(nbC*i)+(nbC+1))] for i in range(nbL)]
```
-
```
def creerMatrice2(nbL,nbC):
    res =[]
    cpt = 1
    for lg in range(nbL):
        res.append([])
        for cl in range(nbC):
            res[lg].append(cpt)
            cpt+=1
    return res
```
-
```
if __name__ == "__main__":
    m = creerMatrice(3,5)
    m2 = creerMatrice2(3,5)
    print(m)
    print(m2)
```

```
>>> %Run comprehension.py
[[1, 2, 3, 4, 5], [6, 7, 8, 9, 10], [11, 12, 13, 14, 15]]
[[1, 2, 3, 4, 5], [6, 7, 8, 9, 10], [11, 12, 13, 14, 15]]
>>>
```

Matrice identité

- Matrice carrée composée que de 0 sauf sur la diagonale sur laquelle il n'y a que des 1

```
def f(x,y):
    if(x==y):
        return 1
    else : return 0

def identity(taille):
    return [ [ f(l,c) for l in range(taille) ] for c in range(taille)]

if __name__ == "__main__":
    i = identity(3)
    print(i)
```

```
>>> %Run comprehension.py
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
>>>
```

Transposée d'une matrice

```
def transpose(m):
    return [ [ligne[i] for ligne in m] for i in range(len(m[0]))]
```

```
if __name__ == "__main__":
    m = creerMatrice(3,4)
    print(m)
    mt = transpose(m)
    print(mt)
```

```
>>> %Run comprehension.py
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
>>>
```

Les PILES et les FILES

CHAPITRE 4

Introduction

Comme toute structure de données dynamique les piles et les files permettent de stocker des éléments. À ces structures de données sont associées des opérations. Pour les files et les piles 2 opérations sont essentielles:

- l'**ajout** d'un élément
- le **retrait** d'un élément

Généralement, les piles et les files auront une capacité limitée. Au-delà d'un certain nombre d'éléments déjà présent dans la structure, il n'est pas possible d'en ajouter de nouveau.

Deux états sont généralement associés à ces structures:

- structure **pleine**
- structure **vide**

La pile

Définition :

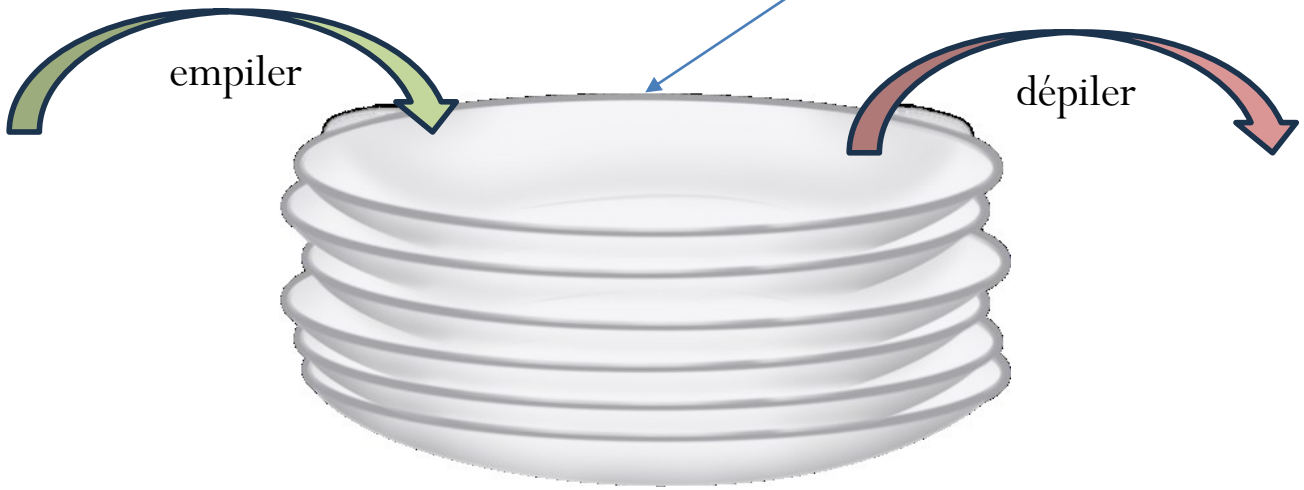
Une pile (en anglais : **stack**) est une structure de données reposant sur le principe de "dernier arrivé, premier sorti".

On parle également de mode LIFO : Last In, First Out.
Ou de mode FILO : First In, Last Out

Analogie :

La première assiette insérée dans la pile est celle qui est en dessous de toutes les autres. La première qui sera retirée de la pile sera celle qui est tout au dessus, donc la dernière déposée.

Sommet de la pile :
C'est la dernière insérée, et
c'est la première qui sera prise



Comment gérer une pile avec Python?

- Utiliser une structure de données déjà existante et l'adapter pour une pile : le type **list**
- Créer sa propre structure de donnée

Utiliser une **list** pour gérer une Pile v1

```
def empiler(p:list,v)->list:
    return [v]+p

def depiler(p:list)->tuple:
    v = p[0]
    nouv = p[1:]
    return (v,nouv)

def estVide(p:list)->bool:
    return len(p)==0

maPile =[]
maPile=empiler(maPile,1)
maPile=empiler(maPile,2)
maPile=empiler(maPile,3)
print(maPile)
valeur,maPile=depiler(maPile)
print("valeur = ",valeur, "mapile =", maPile)
```

⇐ Il faut récupérer la nouvelle liste modifiée

⇐ Il faut récupérer la nouvelle liste modifiée ET la valeur

Utiliser une **list** pour gérer une Pile v2

```
def empiler(p:list,v):
    p.append(v)

def depiler(p:list):
    v = p.pop()
    return v

def estVide(p:list)->bool:
    return len(p)==0

maPile = []
empiler(maPile,1)
empiler(maPile,2)
empiler(maPile,3)
print(maPile)
valeur=depiler(maPile)
print("valeur = ",valeur, "mapile =", maPile)
```

Le sommet de la pile est à la fin de la **list**

Avantages/inconvénients

- Facilité de mise en œuvre

- Pas de type spécifique, utilisation comme une **list** → altération du mécanisme de la pile si pas de précaution
- Pas de gestion de capacité limitée de la pile

Pile en Objet (1)

```
class Element:
    def __init__(self,v,s=None):
        self.valeur = v
        self.suivant = s


class Pile:
    def __init__(self, cap=10):
        self.tete = None
        self.nb = 0
        self.capacite = cap
```

Structure très proche de la liste chaînée

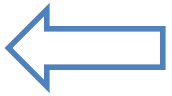
Pile en Objet (2)

```
def estPleine(p:Pile)->bool:
    return p.nb==p.capacite

def estVide(p:Pile)->bool:
    return p.nb==0
```

Gestion de la capacité possible

Pile en Objet (3)

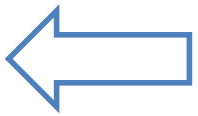
```
def empiler(p:Pile, v):
    if estPleine(p):
        raise Exception("Pile pleine")
    else:
        p.tete=Element(v,p.tete)
        p.nb= p.nb+1

def depiler(p:Pile):
    if estVide(p):
        raise Exception("Pile vide")
    else:
        v = p.tete.valeur
        p.tete = p.tete.suivant
        p.nb = p.nb-1
        return v
```



Gestion d'exceptions possible

Pile en Objet (4)

```
def afficherPile(p:Pile):
    print("[ ",end="")
    elt = p.tete
    while elt!=None :
        print(elt.valeur,end="")
        elt = elt.suivant
        if(elt!=None):
            print(", ",end="")
    print("]")

maPile = Pile(10)
empiler(maPile,1)
empiler(maPile,2)
empiler(maPile,3)
afficherPile(maPile)
valeur=depiler(maPile)
print("valeur = ",valeur, "mapile =")
afficherPile(maPile)
```



Pas de gros changements dans le mode d'emplois

Les Files

Définition :

En informatique, une **file** dite aussi **file d'attente** (en anglais **queue**), est une structure de données basée sur le principe du *premier entré, premier sorti* (en anglais FIFO (« *First In, First Out* ») ou LILO (« *Last In, Last Out* ») ce qui veut dire que les premiers éléments ajoutés à la file seront les premiers à en être retirés.

Analogie

La file d'attente à l'embarquement d'avion, en caisse, etc…

Utiliser une **list** pour gérer une File

```
def enfiler(f:list,v):
    f.append(v)

def defiler(f:list):
    return f.pop(0)

def estVide(f:list)->bool:
    return len(f)==0

maFile =[]
enfiler(maFile,1)
enfiler(maFile,2)
enfiler(maFile,3)
print(maFile)
valeur=defiler(maFile)
print("valeur = ",valeur, "maFile =", maFile)
```

File en objet (1)

```
class Element:
    def __init__(self,v,s=None):
        self.valeur = v
        self.suivant = s


class File:
    def __init__(self, cap=10):
        self.tete = None
        self.queue = None
        self.nb = 0
        self.capacite = cap
```

On enfile (insertion) en queue de liste et on défile (retrait) en tête de liste

File en objet (2)

```
def estPleine(f:File)->bool:
    return f.nb==f.capacite

def estVide(f:File)->bool:
    return f.nb==0
```

File en objet (3)

```
def enfiler(f:File, v):
    if estPleine(f):
        raise Exception("File pleine")
    else:
        nouv = Element(v,None)
        if estVide(f):
            f.queue=nouv
            f.tete = f.queue
        else:
            f.queue.suivant = nouv
            f.queue = nouv
        f.nb= f.nb+1

def defiler(f:File):
    if estVide(f):
        raise Exception("File vide")
    else:
        v = f.tete.valeur
        f.tete = f.tete.suivant
        f.nb = f.nb-1
        if estVide(f):
            queue = None
        return v
```

File en Objet (4)

```
def afficherFile(f:File):
    print("[ ",end="")
    elt = f.tete
    while elt!=None :
        print(elt.valeur,end="")
        elt = elt.suivant
        if(elt!=None):
            print(", ",end="")
    print("]")

maFile = File(10)
enfiler(maFile,1)
enfiler(maFile,2)
enfiler(maFile,3)
afficherFile(maFile)
valeur=defiler(maFile)
print("valeur = ",valeur, "maFile =")
afficherFile(maFile)
```

Remarques

Que ce soit pour les piles ou les files, il existe plusieurs façons pour les implémenter. Dans des langages comme le C, on pourra utiliser des tableaux dont on aura alloué la taille en fonction de la capacité souhaitée.

Pour l'utilisateur, le mode d'emploi est le même. Il n'a pas a connaître le fonctionnement et la structure interne retenue.

Pile et File sont ce qu'on appelle des Types Abstraits de Données (TAD)

Introduction à la complexité algorithmique

CHAPITRE 5

Définition

La complexité d'un algorithme est le nombre d'opérations élémentaires qu'il doit effectuer pour mener à bien un calcul en fonction de la taille des données d'entrée

Notion de performance

```
def dernierElt(uneListe:list):
    return uneListe[-1]

def dernierElt2(uneListe:list):
    for i in range(len(uneListe)):
        if i == len(uneListe)-1:
            res = uneListe[i]
    return res
```

Laquelle des 2 versions est la plus performante?

```
>>> liste = [x**2 for x in range(100)]
>>> 99**2
9801
>>> print(liste)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256,
289, 324, 361, 400, 441, 484, 529, 576, 625, 676, 729, 784, 841, 900,
961, 1024, 1089, 1156, 1225, 1296, 1369, 1444, 1521, 1600, 1681,
1764, 1849, 1936, 2025, 2116, 2209, 2304, 2401, 2500, 2601, 2704,
2809, 2916, 3025, 3136, 3249, 3364, 3481, 3600, 3721, 3844, 3969,
4096, 4225, 4356, 4489, 4624, 4761, 4900, 5041, 5184, 5329, 5476,
5625, 5776, 5929, 6084, 6241, 6400, 6561, 6724, 6889, 7056, 7225,
7396, 7569, 7744, 7921, 8100, 8281, 8464, 8649, 8836, 9025, 9216,
9409, 9604, 9801]
>>> dernierElt(liste)
9801
>>> dernierElt2(liste)
9801
```

Idée : compter le nombre d'actions élémentaires effectuées

Qu'est-ce qu'une action élémentaire?

Affectation, test, opération arithmétique

```
def dernierElt2(uneListe:list):
    for i in range(len(uneListe)):
        if i == len(uneListe)-1:
            res = uneListe[i]
    return res
```

la liste contient **n** éléments
n fois
2 opérations

coût total 2*n

Notion d'ordre de grandeur

- Soit C le coût en actions en fonction de la taille n du problème:

Complexité croissante

Coût	Ordre de grandeur	Exemple d'ordre de comparaison pour n=10	Exemple d'ordre de comparaison pour n=100
c (constante)	**O(1)**	1	1
log(a*n) +c	**O(Log(n))**	3	7
k***n**	**O(n)**	10	100
k***n*log**(a*n)+c	**O(n.Log(n))**	30	700
P(n) de degré 2	**O(n²)**	100	10000
$2^{a*n}+c$	**O(2^n)**	1024	1,26765E+30
(a*n)! + c	**O(n!)**	3628800	9.332622e+157

Exemples d'arithmétique des Ordres de Grandeur:

$O(1)+O(n) = O(n)$

$n * O(1) = O(n)$

$n*O(n) = 0(n^2)$

$O(n)+O(n^2) = O(n^2)$

Opération sur les ***list*** en Python

Operation	Average Case
Copy	O(n)
Append	O(1)
pop(-1)	O(1)
pop(0)	O(n)
Insert	O(n)
Get Item	O(1)
Set Item	O(1)
Delete Item	O(n)
Iteration	O(n)
Sort	O(n log n)
Multiply	O(nk)
x in s	O(n)
min(s), max(s)	O(n)
Get Length	O(1)

Nécessite de décaler

Nécessite de décaler

Nécessite de décaler

Les ***list*** en Python sont implémentées par des tableaux (au sens du C, éléments mis les uns à la suite des autres dans la mémoire).
L'accès à un élément est instantané : l'adresse du $i^{ème}$ élément est à l'adresse de début + i*la taille d'un élément. Cf. cours S2 en C

https://wiki.python.org/moin/TimeComplexity

Comparaison des deux versions

```
def dernierElt2(uneListe:list):
    for i in range(len(uneListe)):
        if i == len(uneListe)-1:
            res = uneListe[i]
    return res
```

la liste contient **n** éléments

n*O(1) = O(n) (2)

O(1) (1)

coût total O(n)

```
def dernierElt(uneListe:list):
    return uneListe[-1]
```

la liste contient **n** éléments

O(1) (1)

coût total O(1)

Attention aux évidences

- Soient deux ***list***, lst1 et lst2, de même taille.
- Quel est le coût de :

```
if lst1 == lst2
```

➔ **O(n) : il faut parcourir tous les éléments et les comparer un à un!**

Etudes de cas

```
mesChampions = {
    #nom    : (defense, attaque, liste des rôles)
    "Teemo" : (3, 5, ["Tireur", "Assassin"]),
    "Jax"   : (5, 7, ["Combattant", "Assassin"]),
    "Elise" : (5, 6, ["Mage", "Combattant"]),
    "Darius": (5, 9, ["Combattant", "Tank"])
}
```

Question : combien ai-je de combattants?

Dénombrer les combattants

```
def denombrer(lesChampions,role):
    cpt = 0
    for champ, caracteristiques in lesChampions.items():
        _,_,roles = caracteristiques
        if role in roles:
            cpt+=1
    return cpt
```

On suppose **n** champions

2 : n x O(1) ➔ **O(n)**

1 : **O(1)**, si on admet qu'il n'y a que 2 ou 3 rôles par champion

```
print("il y a", denombrer(mesChampions, "Combattant"), "combattants")
```

Mesurer le temps

Approche :

- Regarder sa montre avant de démarrer l'algo, puis regarder sa montre après l'exécution et calculer la durée

En Python pour obtenir l'heure actuelle on utilise la fonction time() de la bibliothèque time.

```
import time
print(time.time())
```

`time.time()` renvoie le nombre secondes écoulées depuis le 1/01/1970…

Principe : prendre l'heure actuelle, lancer l'algorithme dont on veut mesurer le temps, et juste après reprendre l'heure actuelle et calculer la durée.

Application à la comparaison de 3 algorithmes de tri

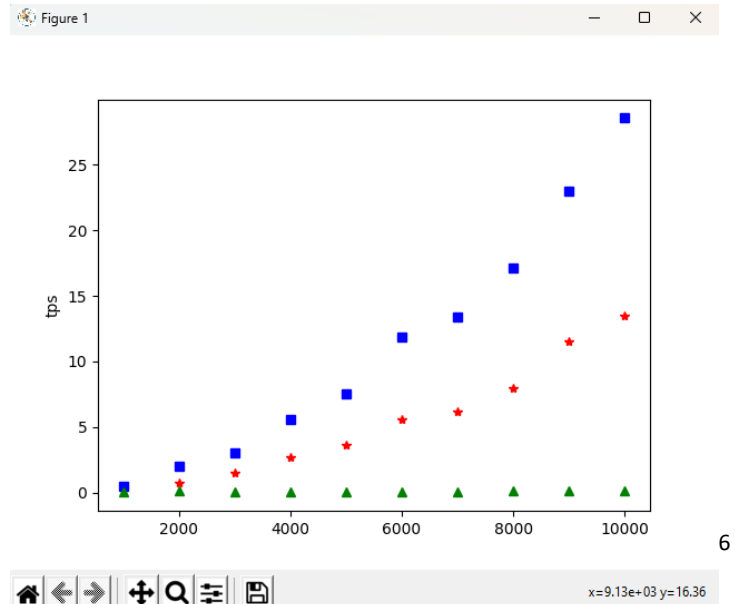
Nous avons vu en cours ou en travaux dirigés 3 algorithmes de tri : Insertion, Sélection et Fusion.

Mode opératoire prendre une liste de n valeurs et la trier avec les 3 algorithmes.

A chaque tour on augmente n.

A la fin on peut tracer pour chaque algorithme de tri le temps mis en fonction de n.

```
def comparer():
    taille = 1000
    tpsInsertion = []
    tpsSelection = []
    tpsFusion = []
    tpsSort = []
    tailles=[]
    while(taille<=10000):
        print("taille = ",taille)
        tailles.append(taille)
        tabInitial = generateTab(taille)
        tab2= tabInitial.copy()
        start = time.time()
        triInsertion(tab2)
        end = time.time()
        tpsInsertion.append( end-start)
        tab2= tabInitial.copy()
        start = time.time()
        triSelection(tab2)
        end = time.time()
        tpsSelection.append( end-start)
        tab2= tabInitial.copy()
        start = time.time()
        triFusion(tab2)
        end = time.time()
        tpsFusion.append( end-start)
        taille=taille+1000

plt.plot(tailles,tpsInsertion,'r*',tailles,tpsSelection,'bs',tailles,tpsFusion,'g^')
    plt.ylabel('tps')
    plt.show()
```



Quelques exemples de calcul de complexité

```
def recherche (element, liste):
        trouve = False
        i = 0
        while not trouve and i<len(liste):
                if liste [i] == element:
                        trouve = True
                else :
                        i = i+1
        if trouve :
                return i
        else :
                return -1
```

Pire des cas : element est à la fin de la liste:

Nombre d'opérations proportionnel à n (len(liste))

0(n)

Quelques exemples de calcul de complexité

```
def dicho(L,x) :
    debut, fin = 0, len(L)-1
    while debut <= fin :
        milieu=(debut+fin)//2
        if x == L[milieu] :
                return milieu
        elif x < L[milieu] :
                fin = milieu-1
        else :
                debut = milieu+1
    return -1
```

Principe « diviser pour régner » : on réduit à chaque fois **la taille** de la partition dans laquelle x pourrait être

Le pire des cas : x est dans L et on fait le maximum de « division », soit k ce nombre

Si n est le nombre d'éléments, on fera combien de divisions?

On s'arrête quand **la taille** est égale à 1

Au commencement taille = n

$1^{ère}$ itération : taille = $n/2$

$2^{ème}$ itération : taille = $n/4$

$k^{ème}$ itération : taille = $n/(2^k) = 1$

➔

$$1 = \frac{n}{2^k}$$

$$2^k = n$$

$$\log_2 2^k = \log_2 n$$

$$k = \log_2 n$$

O(log(n))

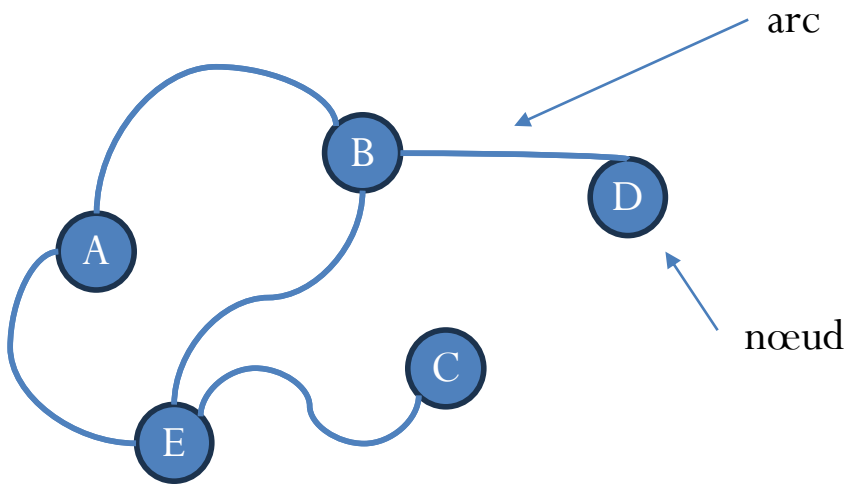
Université Polytechnique HAUTS-DE-FRANCE

Les graphes

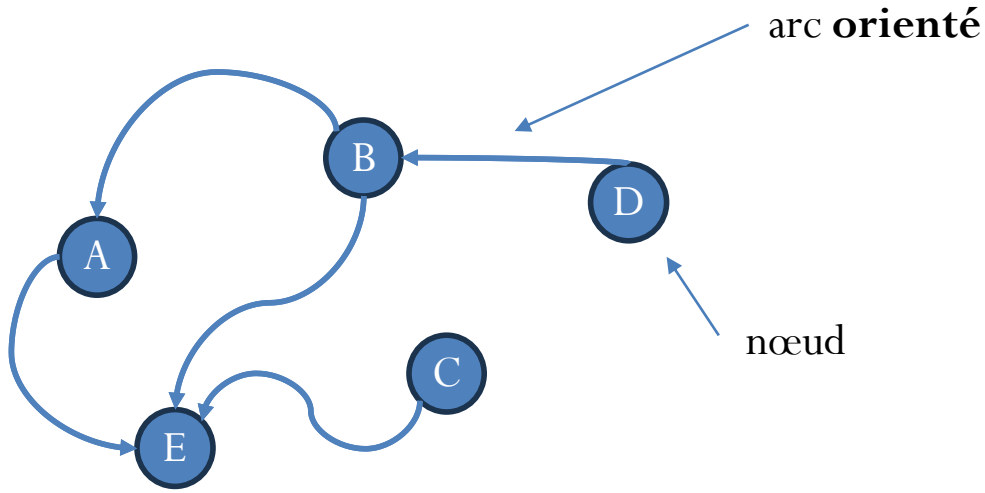
CHAPITRE 6

Définitions

Un **graphe** est un ensemble de **sommet**s reliés entre eux par des **arc**s

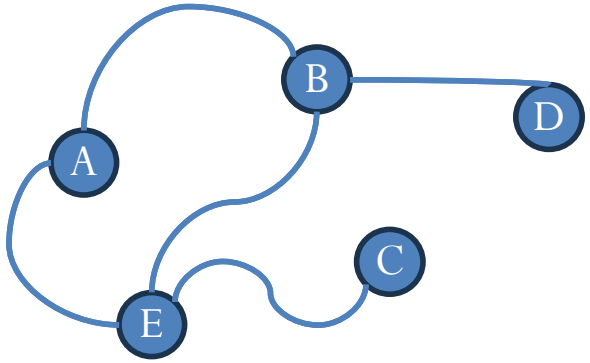


Graphe non orienté



Graphe orienté

Vocabulaire



Voisinage : Quand il y a un arc d'un sommet *s* vers un sommet *t* on dit que *t* est **adjacent** à *s*.
Les sommets adjacents à *s* sont également appelés les **voisins** de *s*

Chemin : Dans un graphe donné, un chemin reliant un sommet *u* à un sommet *v* est une séquence finie de sommets reliés deux à deux par des arcs et menant de *u* à *v*.
Par exemple A→B→E→C est un chemin menant de A à C

Modélisation mémoire d'un graphe

- Par **une matrice d'adjacence** : matrice (A) de NxN pour un graphe de N sommets si $a_{i,j} = 1\ (ou\ True)$ cela signifie qu'il y a un arc (la valeur numérique peut permettre d'associer des poids aux arcs)
  - **Avantage** : facile à mettre en œuvre facilite certains calculs grâce au calcul matriciel
  - **Inconvénient** : pour un graphe avec beaucoup de sommets et peu d'arcs: beaucoup d'espace mémoire consommé « inutilement »

- Par **un dictionnaire** : chaque sommet est une clé à laquelle on associe une liste de sommets voisins
  - **Avantage**: accès rapide, consommation mémoire ajustée
  - **Inconvénient** : moins de souplesse de calcul

Modélisation par matrice d'adjacence

```
class Graphe:
    def __init__(self,nbsommets):
        self.adj = [[0]*nbsommets for _ in range(nbsommets)]
        self.labels = []
        self.dico =dict()
        for  i in range(len(self.adj[0])):
            self.labels.append(str(i))
            self.dico[str(i)]=i

    def setLabels(self,etiquettes):
        if len(etiquettes)!=len(self.adj[0]):
            raise Exception("taille incompatible")
        self.labels=[]
        for  i in range(len(self.adj[0])):
            self.labels.append(etiquettes[i])
            self.dico[etiquettes[i]]=i

    def ajoutArc(self, s1, s2):
        self.adj[self.dico[s1]][self.dico[s2]] = 1
        self.adj[self.dico[s2]][self.dico[s1]] = 1
```

labels : permet d'associer une étiquette à un nœud

dico : permet d'associer un indice à une étiquette

```
    def voisins(self, s):
        vois = []
        for col in range (len(self.adj[0])):
            if self.adj[self.dico[s]][col] !=0:
                vois.append(self.labels[col])
        return vois

def __str__(self):
        # affichage des etiquettes
        ch=""
        for e in self.labels:
            ch = ch+"\t"+e
        ch=ch+"\n«
        # affichage de la matrice
        for lig in range(len(self.adj[0])):
            ch = ch+self.labels[lig]+'\t'
            for col in range(len(self.adj[0])):
                ch=ch+str(self.adj[lig][col])+'\t'
            ch=ch+'\n'
        return ch
```

Modélisation par matrice d'adjacence

```
42  graph = Graphe(5)
43  graph.setLabels(["A","B","C","D","E"])
44  graph.ajoutArc("A","B")
45  graph.ajoutArc("A","E")
46  graph.ajoutArc("B","D")
47  graph.ajoutArc("B","E")
48  graph.ajoutArc("E","C")
49  print(graph)
50  print(graph.voisins("E"))
51
52
53
```

Shell | Exception | Program tree

```
>>> %Run grapheAdj.py
      A     B     C     D     E
A     0     1     0     0     1
B     1     0     0     1     1
C     0     0     0     0     1
D     0     1     0     0     0
E     1     1     1     0     0

['A', 'B', 'C']
```

Affichage de la matrice d'adjacence

La liste des voisins de « E »

Modélisation par dictionnaire

```python
class Graphe:
    def __init__(self):
        self.adj = dict()

    def ajoutSommet(self, s):
        self.adj[s] = []

    def ajoutArc(self,s1,s2):
        if s1 not in self.adj:
            self.adj[s1]=[]
        if s2 not in self.adj:
            self.adj[s2]=[]
        self.adj[s1].append(s2)
        self.adj[s2].append(s1)

    def voisins(self,s):
        return self.adj[s]

    def __str__(self):
        ch = str(self.adj)
        return ch

graph = Graphe()
graph.ajoutArc("A","B")
graph.ajoutArc("A","E")
graph.ajoutArc("B","D")
graph.ajoutArc("B","E")
graph.ajoutArc("E","C")
print(graph)
print(graph.voisins("E"))
```

Permet d'ajouter les sommets s'ils ne sont pas encore créés

Shell | Exception | Program tree

```
>>> %Run grapheDic.py
 {'A': ['B', 'E'], 'B': ['A', 'D', 'E'], 'E': ['A', 'B', 'C'], 'D': ['B'], '
C': ['E']}
['A', 'B', 'C']
```

Introduction à GraphViz

GraphViz est un logiciel opensource pour la visualisation de graphes.
Le logiciel peut être téléchargé à cette adresse : https://www.graphviz.org/

Un graphe y est décrit selon un formalisme textuel et permet la visualisation de ce graphe notamment en générant un pdf.

Il existe un module python graphviz qui permet de générer en python une description d'un graphe en langage Graphviz et de demander à GraphViz de générer le pdf.
Pour cela il faut avoir installé le logiciel GraphViz sur votre machine et d'installer (via pip par exemple) le module Python GraphViz.

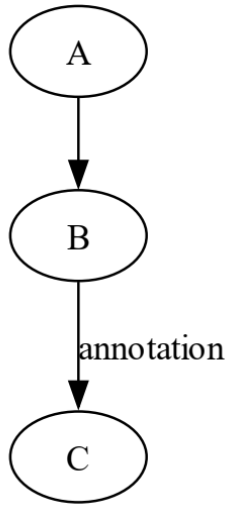
Exemple d'utilisation du module GraphViz

```
from graphviz import Digraph
import os

os.environ['PATH'] += os.pathsep +"C:\Program
Files\Graphviz\bin"

g = Digraph()
g.node("A")
g.node("B")
g.node("C")
g.edge("A","B")
g.edge("B","C",'annotation')

g.render("exemple", view=True)
```

Chemin où sont installés les exécutables de GraphViz

Description du graphe

Pdf généré



Exploitation pour visualiser notre graphe

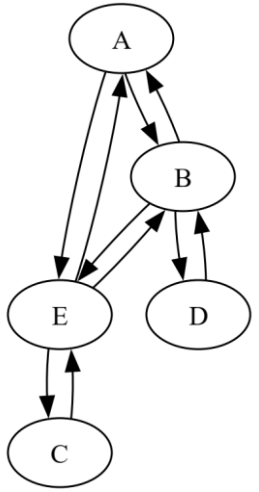
```
def toImage(self):
        graphe=Digraph()
        for s in self.adj:
            graphe.node(s)
        for s in self.adj:
            for v in self.voisins(s):
                graphe.edge(s,v)
        graphe.render("graphe", view = True)
```

Ajout d'une méthode

Chaque clé du dictionnaire est un nœud qu'on ajoute au graphe

On crée un arc entre chaque nœud et chacun de ses voisins

```
graph = Graphe()
graph.ajoutArc("A","B")
graph.ajoutArc("A","E")
graph.ajoutArc("B","D")
graph.ajoutArc("B","E")
graph.ajoutArc("E","C")
print(graph)
print(graph.voisins("E"))
graph.toImage()
```

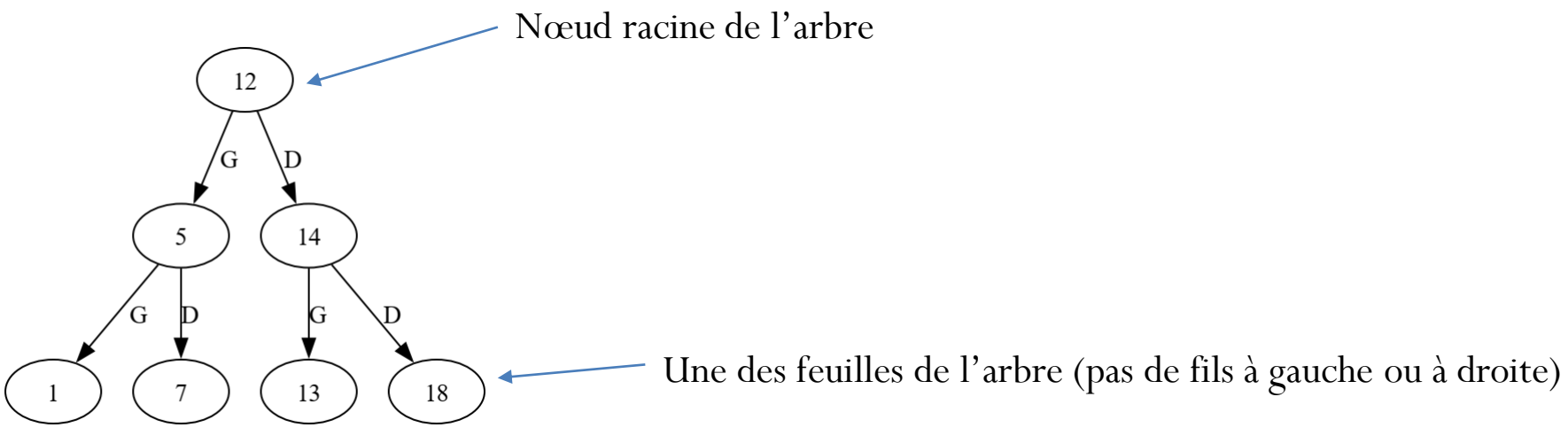


Les Arbres Binaires de Recherche (ABR)

Un Arbre Binaire de Recherche (ABR) est un Type Abstrait de Données (TAD) qui permet de gérer une collection d'éléments qu'on ordonne (il doit donc y avoir la possibilité de comparer des éléments entre eux).

Un arbre se compose de nœuds, ou chaque nœud est associé à une valeur et au plus 2 nœuds fils.
Par convention, on place à gauche les éléments plus petits et à droite les autres.

Un ABR est défini par son nœud racine.

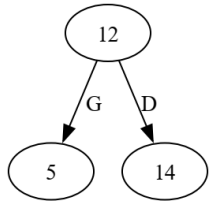


Ajout d'une valeur dans un ABR

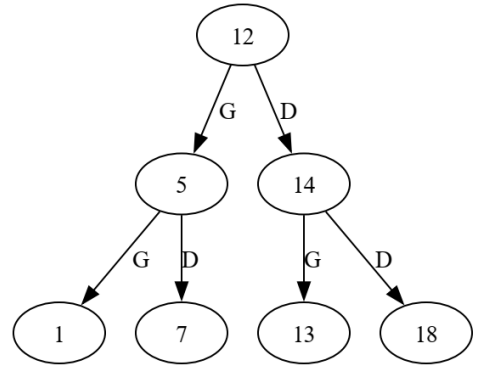
```
a = ABR()
```

Arbre vide

```
a = ABR()
a.ajouter(12)
a.ajouter(14)
a.ajouter(5)
```




```
a = ABR()
a.ajouter(12)
a.ajouter(14)
a.ajouter(5)
a.ajouter(13)
a.ajouter(7)
a.ajouter(1)
a.ajouter(18)
```



Parcours d'un ABR

On commence par le nœud racine:

Arrivé sur un nœud :

- Parcours préfixé : on lit la valeur nœud, puis celles des fils à gauche et enfin celles des fils à droite
- Parcours postfixé: on lit la valeur des nœuds des fils à gauche, puis celles des fils à droite et enfin la valeur du nœud
- Parcours infixé : on lit la valeur des nœuds des fils à gauche, puis la valeur du nœud et enfin celles des fils à droite

Le parcours infixé d'un ABR permet de parcourir les valeurs de l'arbre dans l'ordre!

Les opérations sur un ABR

- Ajout d'une valeur
- Suppression d'une valeur
- Recherche d'une valeur
- Parcours des valeurs (par exemple affichage, ou création d'une liste triée à partir des valeurs contenues dans l'ABR)

Université Polytechnique HAUTS-DE-FRANCE

Aller plus loin en programmation orienté objet en Java

CHAPITRE 7

La méthode __str__()

```
1  class Point:
2      def __init__(self, vx=0.0, vy=0.0):
3          self.__x = vx
4          self.__y = vy
5
6  p = Point()
7  print("p=",p)
8
```

Console

```
>>> %Run exemplePOO.py
p= <__main__.Point object at 0x04313EB0>
>>>
```

Par défaut, affiche une chaîne de caractères composé du nom de la classe et de l'adresse mémoire où est stocké l'objet. Notre objet a été convertir en str.

La méthode __str__()

```
1  class Point:
2      def __init__(self, vx=0.0, vy=0.0):
3          self.__x = vx
4          self.__y = vy
5
6      def __str__(self):
7          return "je suis un Point"
8
9  p = Point()
10 print("p=",p)
11 ch = str(p) + " c'est tout!"
12 print(ch)
```

Console

```
>>> %Run exemplePOO.py
p= je suis un Point
je suis un Point c'est tout!
>>>
```

La redéfinition de la méthode **__str__** permet de spécifier la chaîne de caractères qui sera utilisée pour convertir l'objet en **str**

Attributs privés

```python
class Point:
    def __init__(self, vx=0.0, vy=0.0):
        self.__x = vx
        self.__y = vy

    def __str__(self):
        return "("+str(self.__x)+", "+str(self.__y)+")"

p = Point()
print("p=",p)
print("p.__x=",p.__x)
```

Console

```
>>> %Run exemplePOO.py
p= (0.0, 0.0)
Traceback (most recent call last):
  File "C:\Owncloud\DEV\Python\2023-2024\CM\exemplePOO.py", li
    print("p.__x=",p.__x)
AttributeError: 'Point' object has no attribute '__x'
```

Pour rendre un attribut **privé** il faut le préfixer par **2 caractères soulignés** (underscore _) et ne pas le terminer par deux caractères soulignés.

Un attribut privé est inconnu en dehors de la classe.

Intérêt des membres privés

- Masquer la « composition » des objets, l'utilisateur de la classe n'a pas à connaître les détails
- Interdire la possibilité de modifier directement certains attributs ce qui pourrait provoquer des erreurs
- Cadrer les accès (lecture, mais surtout écriture) en passant par des méthodes dédiées

Exemple la classe Pixel

Un Pixel possède 3 attributs :

- Abscisse forcément une valeur entière positive
- Ordonnée: forcément une valeur entière positive
- Couleur : un tuple de 3 entiers forcément compris entre 0 et 255 correspondants aux composantes Rouge, Vert, Bleu

Un Pixel ne peut pas être déplacé

Un Pixel peut changer de couleur

```
class Pixel:
    def __init__(self, vx=0,vy=0, vc =(128,128,128)):
        if type(vc)!=tuple:
            raise Exception("erreur de type pour la couleur tuple attendu")
        if len(vc)!=3:
            raise Exception("erreur de type pour la couleur tuple de 3 int attendu")
        for c in vc:
            if type(c) != int:
                raise Exception("erreur de type pour pour la composante couleur : int attendu")
            if c>255 or c<0:
                print("c=",c)
                raise Exception("valeur pour la composante couleur hors interval  entre 0 et
255")
        if type(vx)!=int:
            raise Exception("erreur de type x int attendu")
        if type(vy)!=int:
            raise Exception("erreur de type y int attendu")
        self.__x = vx
        self.__y = vy
        self.__couleur = vc

    def getX(self):
        return self.__x

    def getY(self):
        return self.__y

    def getColor(self):
        return self.__couleur

    def setColor(self,vc):
        if type(vc)!=tuple:
            raise Exception("erreur de type pour la couleur tuple attendu")
        if len(vc)!=3:
            raise Exception("erreur de type pour la couleur tuple de 3 int attendu")
        for c in vc:
            if type(c) != int:
                raise Exception("erreur de type pour pour la composante couleur : int attendu")
            if c>255 or c<0:
                raise Exception("valeur pour la composante couleur hors interval  entre 0 et
255")
        self.__couleur=vc

    def __str__(self):
        return "Pixel: ("+str(self.__x)+", "+str(self.__y)+") , "+str(self.__couleur)
```

Surcharge des opérateurs

Nous avons vu précédemment des méthodes « magiques » qui sont appelés automatiquement __init__() et __str__().
Il existe d'autres méthodes « magiques » qui permettent de définir le comportement de certains opérateurs

Opérateur	Méthode magique
+	__add__(self, autre)
–	__sub__(self, autre)
*	__mul__(self, autre)
/	__truediv__(self, autre)
//	__floordiv__(self, autre)
%	__mod__(self, autre)
**	__pow__(self, autre)
>>	__rshift__(self, autre)
<<	__lshift__(self, autre)
&	__and__(self, autre)
|	__or__(self, autre)
^	__xor__(self, autre)

Opérateur	Méthode magique
<	__lt__(self, autre)
>	__gt__(self, autre)
<=	__le__(self, autre)
>=	__ge__(self, autre)
==	__eq__(self, autre)
!=	__ne__(self, autre)

Application la classe Complexe

héritage

```
1  class Point:
2      def __init__(self,vx=0,vy=0):
3          self.x=vx
4          self.__y=vy
5
6      def __str__(self):
7          return "({},{})".format(self.x,self.__y)
8
9  class PointColore(Point):
10     def __init__(self,vx=0,vy=0,vc="Noir"):
11         Point.__init__(self,vx,vy)
12         self.couleur = vc
13
14     def __str__(self):
15         return "[{}, {}]".format(Point.__str__(self), self.couleur)
16
17 p=Point()
18 print(p)
19 print(p.__dict__)
20 pc=PointColore()
21 print(pc)
22 print(pc.__dict__)
23 pc._Point__y = 12
24 p._Point__y = 25
25 print(p.__dict__)
26 print(pc.__dict__)
```

Indique que la classe PointColore ***hérite*** de la classe Point

Appel du constructeur de la classe Point

Retourne sous forme de ***dict*** les attributs de l'objet

```
>>> %Run heritage.py

  (0,0)
  {'x': 0, '_Point__y': 0}
  [(0,0), Noir]
  {'x': 0, '_Point__y': 0, 'couleur': 'Noir'}
  {'x': 0, '_Point__y': 25}
  {'x': 0, '_Point__y': 12, 'couleur': 'Noir'}
```

La classe ListeChainee itérable

Cf. code python