Introduction à la programmation avec Python

FULLSTACK – Algo

Xavier Crégut <prenom.nom@enseeiht.fr>

Objectif du module

Apprendre à bien écrire des programmes. . .
. . .dans un style impératif. . .
. . .en s'appuyant sur le langage Python . . .et en utilisant la méthode des raffinages

Quelques recommandations

- Le texte de la forme Python contient un lien, cliquer dessus pour le suivre
- Ce support est conçu pour être lu de manière autonome
- Il est alors conseillé de faire les exercices qui apparaissent dans le support au fur et à mesure
- Pour certains exercices une indication de temps est donnée :
  - inutile d'y passer plus de temps
  - continuer la lecture : des indices ou une réponse sont donnés dans la suite
- Si vous avez du code Python à exécuter, vous pouvez utiliser :
  - une console Python en ligne pour les réponses courtes (1 à 2 instructions) : Python shell
  - un interpréteur de programmes Python avec visualisation de l'exécution : Python tutor
  - un interpréteur Python dans votre navigateur : Repl.it ou Trinket
  - une version de Python installée sur votre machine

Sommaire

- Guider un robot
- Raffinages
- Sous-programmes

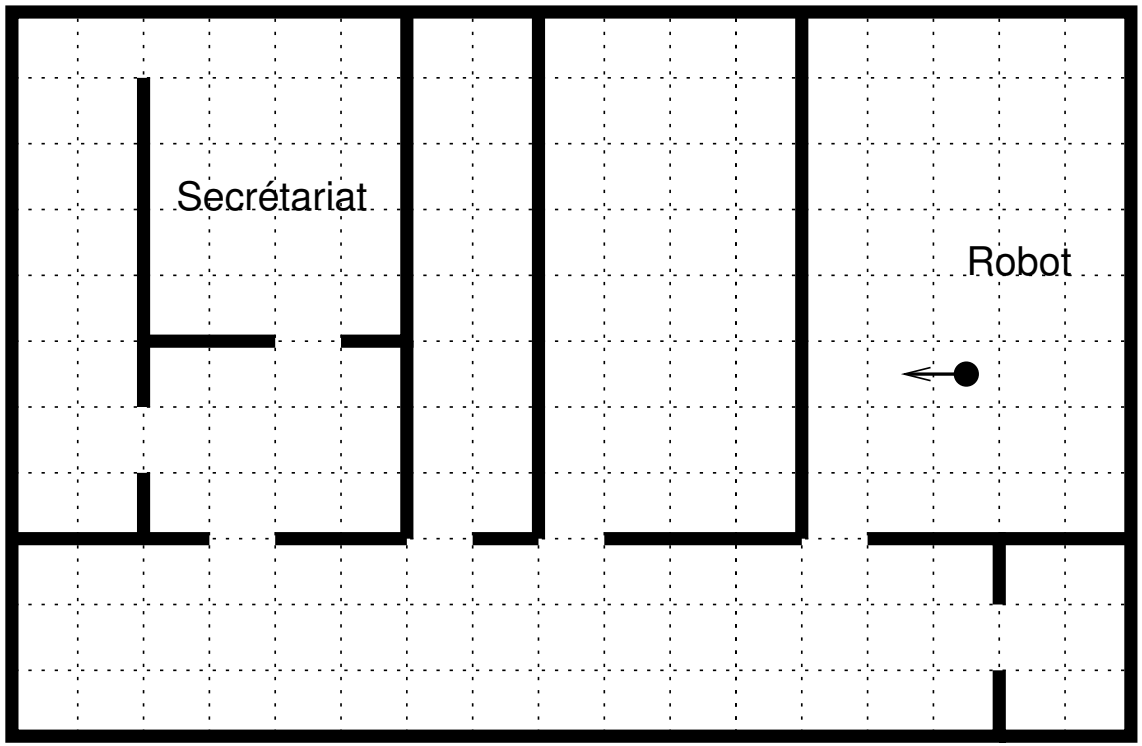
Xavier Crégut <prenom.nom@enseeiht.fr>

Pourquoi ce survol ?

- Comprendre sur un exemple simple les principaux concepts qui seront détaillés dans la suite
- Comprendre la démarche de résolution de problèmes qui sera mise en œuvre
- Comprendre quelques critères importants dans l'écriture de programme
- Entrevoir les principaux concepts pour limiter les références en avant dans la suite

Guider un robot

Exercice (3 minutes) Écrire, dans un format libre, ce que doit faire le robot pour aller de la salle de cours (sa position initiale) jusqu'au secrétariat sachant qu'il sait juste avancer d'une case et pivoter de 90 ° vers la droite.



Exercice (3 minutes) Répondre de manière concise et précise aux questions suivantes :

1. Quelle démarche avez-vous suivie pour résoudre l'exercice précédent ?
2. Expliquez en quelques phrases en français le chemin suivi par le robot.
3. Pourquoi avoir choisi ce chemin ?
4. Est-ce que le problème posé était clair ?

Quelle démarche avez-vous suivie pour résoudre l'exercice ?

Démarche suivie

La réponse dépend de chacun, vos actions ont peut-être consisté en :

- Comprendre ce qui est demandé
- Identifier les capacités robots (les actions qu'il sait faire)
- Choisir un chemin
- Le traduire en actions du robot

Le robot

- On s'adresse au robot : le robot est notre **processeur** ; il va exécuter nos ordres/actions.
- Le robot ne comprend que deux ordres ou actions :
  - avancer d'une case ;
  - pivoter de 90° à droite.
- Ce sont ses **instructions** (actions élémentaires).
- On choisit une manière de les noter : avancer, A... pour avancer ; pivoter, P... pour pivoter

Attention : Bien comprendre l'effet de chaque instruction est essentiel pour savoir bien les utiliser !

Définitions

- **Processeur :** Élément (machine, homme, etc.) qui exécute un programme.
- **Instruction :** Action que sait réaliser le processeur
- **Programme :** Suite d'instructions

Quelques « programmes » proposés

Est-ce vraiment des programmes ? Oui et non : ils répondent au problème posé mais manquent de généralité : ils ne fonctionnent que pour cette configuration des lieux, cette position initiale du robot et ce point d'arrivée ! **Normalement, un programme traite une classe de problèmes.**

Proposition 1 :

```
avancer avancer pivoter pivoter
pivoter avancer avancer avancer
pivoter avancer avancer avancer
avancer avancer avancer avancer
avancer avancer pivoter avancer
avancer avancer pivoter avancer
pivoter pivoter pivoter avancer
```

Proposition 2 :

```
avancer x 2
pivoter x 3
avancer x 3
pivoter
avancer x 9
pivoter
avancer x 3
pivoter
avancer
pivoter x 3
avancer
```

Proposition 3 :

```
avancer
avancer
pivoter
pivoter
pivoter
avancer
avancer
avancer
avancer
pivoter
avancer
avancer
avancer
avancer
avancer
avancer
avancer
pivoter
avancer
pivoter
pivoter
avancer
```

Proposition 4 :

```
A A P P P A A A P A A A A A A A A
A P A A A P A P P P A
```

Exercice (5 minutes) Répondre aux questions suivantes :

1. Est-ce que le robot comprend les programmes proposés ?
2. Quel est le chemin suivi par le robot ?
3. Est-ce qu'un lecteur humain comprend le programme ?
4. Pourquoi faire « pivoter x 3 » ou « pivoter pivoter pivoter » ou « P P P » ?

Xavier Crégut <prenom.nom@enseeiht.fr>

Leçons apprises de cet exercice

Vous savez écrire un programme

C'est une bonne chose ! Mais il faut aussi savoir **bien écrire** le **bon programme**.

Connaître le processeur est essentiel

Le programme est correct s'il n'utilise que des actions comprises par le processeur : ses instructions
Capacités du robot : Le robot ne sait faire que deux choses, n'a que deux instructions :

1. avancer d'une case suivant la direction courante (avancer ou A)
2. pivoter à droite de 90° (pivoter ou P)

Conclusion : Le robot ne comprend pas la proposition 2 (à cause des facteurs multiplicatifs)
Elle reste pratique pour l'humain : lui évite de compter le nombre de mots consécutifs identiques.

Compréhensibilité du programme : le programme doit être facile à lire

1. Mettre une action par ligne : on écarte donc les propositions 1 et 4
2. Comprendre pourquoi les actions ont été écrites : on écarte les propositions 1, 2, 3 et 4 !
   - Voir la question « Pourquoi faire « pivoter x 3 » ou « pivoter pivoter pivoter » ou « P P P » ? »

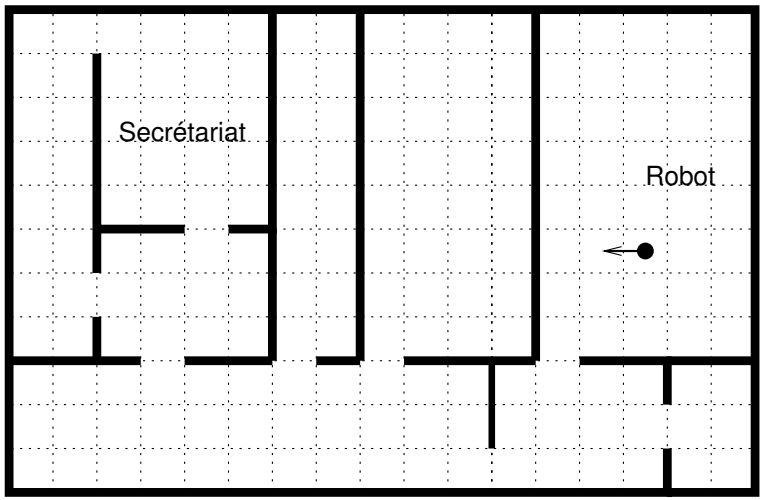
Conclusion

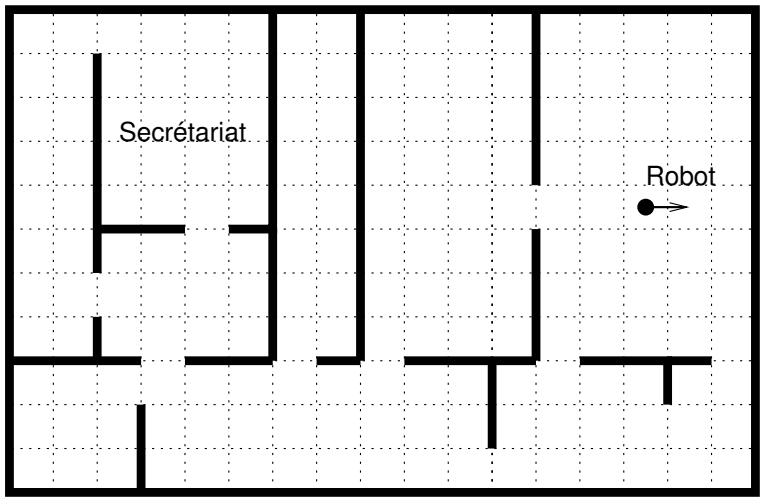
Nous avons écarté toutes les propositions !

Évolutions possibles dans l'énoncé

Exercice (3 minutes) Est-il facile de faire évoluer les programmes précédents pour prendre en compte les modifications suivantes ?

1. Nouvelles configurations des lieux :





2. Changement du robot :
   Le robot tombe en panne. Le nouveau modèle avance d'une case ou pivote à gauche de 90°.

Évolutivité / extensibilité

Un petit changement dans l'énoncé doit se traduire par de petites modifications localisées dans une partie du programme.

Réponses aux différentes questions

Expliquez en quelques phrases en français le chemin suivi par le robot.

C'est la même question que "Quel est le chemin suivi par le robot ?"
C'est à vous de le faire... sans paraphraser les propositions 1 à 4.
Essayez encore (2 minutes)...

Pourquoi avoir choisi ce chemin ?

Certainement parce que c'est le plus court.
Remarque : sur la fin du chemin plusieurs variantes de mêmes longueurs sont possibles.

Est-ce que le problème posé était clair ?

Votre réponse était certainement oui. Mais est-ce si sûr ?
Les besoins sont souvent exprimés incomplètement. Par exemple, faut-il choisir le chemin le plus court/rapide ou celui qui couvre le maximum de surface (robot aspirateur) ?
Essentiel : Être sûr d'avoir bien compris le problème $\Longrightarrow$ **reformuler et prendre des exemples !**

Réponses aux différentes questions

Prendre en compte la nouvelle configuration des lieux (celle de gauche)

- Il faut reprendre tout le début du programme.
- Il n'est pas facile de s'y retrouver. Heureusement qu'il y a la longue série de « avancer ».

Prendre en compte la nouvelle configuration des lieux (celle de droite)

- Il est ici plus difficile de choisir le chemin à suivre.
- Le plus court en nombre de cases consiste à faire faire demi-tour au robot
- Mais il comporte beaucoup plus de « pivoter » que celui qui consiste à partir tout droit
- Impossible de choisir entre les deux sans connaitre le coût des instructions `avancer` et `pivoter`
- **Conclusion :** Pas toujours facile de savoir ce qu'est la version optimale d'un programme
- Ce ne sera pas notre priorité, mais il ne faut pas écrire un programme manifestement inefficace

Changement de robot : le nouveau pivote de 90° à gauche

- Il faut remplacer, dans tout le programme, les « P » par « PPP » et les « PPP » par « P »
- Ce n'est pas très pratique !

Conclusion : Le programme n'est pas évolutif !

Réponses aux différentes questions

Est-ce qu'un lecteur humain comprend le programme ?

Difficilement !
Il est obligé de lire instruction par instruction et de retrouver l'**intention** de l'auteur du programme.
Principe : Le programmeur doit faciliter la vie au lecteur en explicitant son intention !
Jusitification : Le programme devra évoluer pour intégrer de nouveaux besoins...

Pourquoi faire « pivoter x 3 » (ou « pivoter pivoter pivoter » ou « P P P ») ?

Pour tourner à gauche. C'est justement l'intention du programmeur. C'est une action complexe (car non comprise par le processeur). Elle permet au lecteur de comprendre l'intention du programme. Il faut qu'elle reste dans le programme.
Conséquence : On peut écrire un programme en utilisant des actions complexes. Il faudra juste les expliquer au processeur ensuite.
Analogie : C'est comme les lemmes en mathématiques : ils aident à démontrer un théorème mais la démonstration n'est valide que si les lemmes sont eux-mêmes démontrés.

Nouvelles questions (2 minutes)

1. Pourquoi faire 9 fois avancer ?
2. Pourquoi commencer par faire « A A P P P A A A » ?

La question « pourquoi ? » permet de retrouver les actions complexes, l'intention du programmeur.

L'exemple du robot revu : le programmeur explicite ses intentions

R_0 : Reformulation du problème

```
R0 : Guider le robot de la salle de cours vers le secrétariat
```

Solution informelle

Faire suivre au robot le chemin le plus « court ».
(On nomme les salles pour s'y retrouver : couloir, vestibule, etc.)

R_1 : Décomposition de l'action complexe R_0 résumant le programme à écrire

```
R1 : Comment « Guider le robot de la salle de cours vers le secrétariat » ?
       | Sortir de la salle de cours
       | Longer le couloir
       | Traverser le vestibule
```

Principe : C'est une approche de type « diviser pour régner » (*divide and conquer*)

- Chaque action introduite décrit un **nouveau problème à résoudre** (diviser)
- La décomposition les **combine** pour atteindre l'objectif général R_0 (combiner)
- Les actions introduites sont **complexes** : il faut à leur tour les **décomposer** (régner)

C'est la **méthode des raffinages**

Les décompositions continuent

R_2 : décomposition des actions identifiées au niveau R_1

```
R2 : Comment « Sortir de la salle de cours » ?
      | Progresser de 2 cases
      | Tourner à gauche
      | Progresser de 3 cases

R2 : Comment « longer le couloir » ?
      | Tourner à droite
      | Progresser de 9 cases
      | Tourner à droite

R2 : Comment « traverser le vestibule » ?
      | Progresser de 3 cases
      | Tourner à droite
      | Progresser de 1 case
      | Tourner à gauche
      | Progresser de 1 case
```

Les décompositions continuent. . . et se terminent

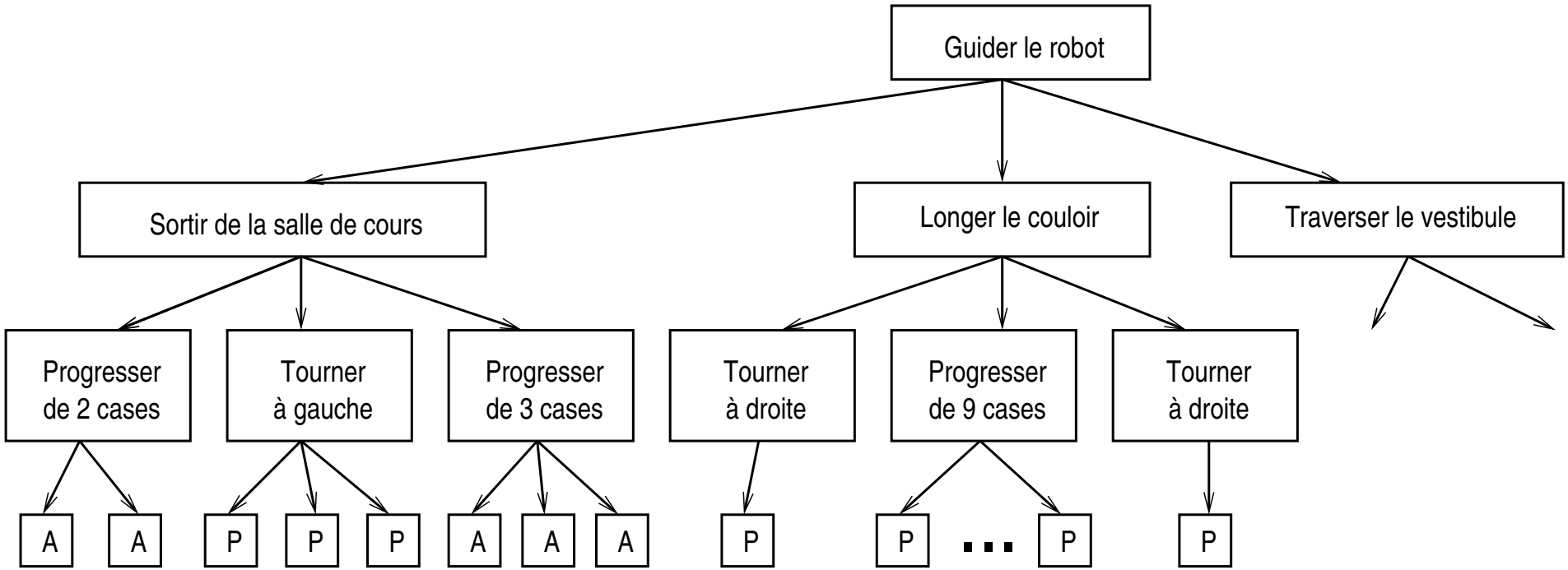
R_3 : décomposition des actions de niveau R_2

```
R3 : Comment « Tourner à gauche » ?
      | pivoter
      | pivoter
      | pivoter

R3 : Comment « Tourner à droite » ?
      | pivoter

R3 : Comment « Progresser de 2 cases » ?
      | avancer
      | avancer

R3 : Comment « Progresser de 3 cases » ?
      | avancer
      | avancer
      | avancer

R3 : Comment « Progresser de 9 cases » ?
      | avancer
      | avancer
      | avancer
      | avancer
      | avancer
      | avancer
      | avancer
      | avancer
      | avancer
```

Quand arrête-t-on ?

Quand toutes les actions complexes ont été décomposées.
Les derniers niveaux de raffinage n'utilisent donc que des instructions du processeur.
Avec l'expérience, on peut s'arrêter un peu plus tôt (action complexe connue. . .).

Les décompositions sous forme d'arbre



Remarque : Peu pratique quand des structures de contrôle autre que la séquence sont utilisées

Obtention (automatique) du programme

Principe

1. Parcours préfixe de l'arbre : en profondeur, de gauche à droite.
2. Chaque action complexe devient un commentaire (explique l'intention du programmeur)

Le programme

```
def guider_robot():
    '''
    Guider le robot de la salle
    de cours vers le secrétariat
    '''
    # Sortir de la salle de cours
    #   Progresser de 2 cases
    avancer
    avancer
    #   Tourner à gauche
    pivoter
    pivoter
    pivoter
    #   Progresser de 3 cases
    avancer
    avancer
    avancer
    # Longer le couloir
    #   Tourner à droite
    pivoter
    #   Progresser de 9 cases
    avancer
    avancer
    avancer
    avancer
    avancer
    avancer
    avancer
    avancer
    avancer

    # Traverser le vestibule
    #   Tourner à droite
    pivoter
    #   Progresser de 3 cases
    avancer
    avancer
    avancer
    #   Tourner à droite
    pivoter
    #   Progresser de 1 case
    avancer
    #   Tourner à gauche
    pivoter
    pivoter
    pivoter
    #   Progresser de 1 case
    avancer
```

Le programme est identique à la proposition 3, les commentaires en plus.
L'intention du programmeur est donc explicite.

Sous-programmes

Constat

- Dans le programme précédent, utiliser plusieurs fois « tourner à gauche » a des défauts
  - le code est redondant : on trouve plusieurs fois « pivoter pivoter pivoter »
  - s'il y a une erreur dans ce code, il faudra corriger toutes les occurrences
  - limite l'évolution du code : cas où le robot ne sait que pivoter de 90° à gauche
- Toujours éviter les redondances ! Un outil : le **sous-programme**

Sous-programme

- Un sous-programme est le moyen pour le programmeur de définir de nouvelles « instructions »
- et de les utiliser plusieurs fois
- dans ce programme et dans d'autres programmes (**réutilisation**)

Exemple : sous-programmes touner_gauche et tourner_droit

```
1 def tourner_gauche():
2     '''Tourner à gauche.'''
3     pivoter
4     pivoter
5     pivoter
```

```
1 def tourner_droite():
2     '''Tourner à droite.'''
3     pivoter
```

Sous-programme avec paramètre

Constat

- Dans le programme précédent, on fait avancer le robot de 2 cases, 3 cases, 1 case ou 9 cases
- Ce qui change c'est le nombre de cases, on peut **en faire un paramètre**
- le sous-programme `progresser` consiste à faire avancer le robot de *n* cases

Sous-programme : Définition

```
1 def progresser(n: int):
2     '''Avancer de n cases (n > 0).'''
3     for i in range(n):    # n fois
4         avancer
```

Utilisation

```
progresser(2)   # appel à progresser avec n valant 2
tourner_gauche()
progresser(3)   # appel à progresser avec n valant 3
...
```

Paramètre formel et paramètre effectif

- **Paramètre formel** dans la définition du sous-programme
  - on ne connaît pas sa valeur
  - on sait juste que c'est un entier > 0 (cf documentation)
- **Paramètre effectif** lors de l'utilisation du sous-programme
  - donne une valeur au paramètre formel
  - le sous-programme s'exécute avec une valeur de *n* connue
- **Analogie avec les fonctions en math**
  - $f(x) = 2x + 3$
  - $y0 = f(0)$

analogie avec les fonctions en math

```
def f(x: float) -> float:
    #! x paramètre formel
    #! étant donné x,
    #! le résultat de
    #! la fonction est :
    return 2 * x + 3

y0 = f(0);
    #! 0 paramètre effectif
```

Guider le robot en utilisant les sous-programmes

```
1
2  def tourner_gauche():
3      '''Tourner à gauche.'''
4      pivoter
5      pivoter
6      pivoter
7
8  def tourner_droite():
9      '''Tourner à droite.'''
10     pivoter
11
12 def progresser(n: int):
13     '''Avancer de n cases (n > 0).'''
14     for i in range(n):    # n fois
15         avancer
```

Bonnes propriétés :

- Pas de redondance
- Programme découpé en sous-programmes donc :
  - plus facile à comprendre
  - plus facile à faire évoluer
- Sous-programmes courts avec :
  - un objectif bien identifié (premières lignes)
  - des instructions qui réalisent cet objectif
- Le programme est un sous-programme sans paramètre
- Rq: Les actions complexes du R1 auraient pu être des SP

```
15 def guider_robot():
16     '''
17     Guider le robot de la salle
18     de cours vers le secrétariat
19     '''
20     # Sortir de la salle de cours
21     progresser(2)
22     tourner_gauche()
23     progresser(3)
24
25     # Longer le couloir
26     tourner_droite()
27     progresser(9)
28
29     # Traverser le vestibule
30     tourner_droite()
31     progresser(3)
32     tourner_droite()
33     progresser(1)
34     tourner_gauche()
35     progresser(1)
```

Xavier Crégut <prenom.nom@enseeiht.fr>

Bilan

Cet exemple introductif nous a permis de voir sur l'exemple du robot ce que nous allons faire

Un processeur : Le « langage Python » (en fait son interpréteur)

- les instructions élémentaires : l'équivalent de `avancer` et `pivoter`
- les données élémentaires : le nombre de cases (entier), etc.
- les structures de contrôle : par exemple, `for` pour contrôler les instructions
- les données structurées : liste, par exemple pour représenter le lieu où évolue le robot

Une méthode : la méthode des raffinages

- construire progressivement une solution algorithmique
- traiter des problèmes de petite et grande taille
- expliciter l'intention du programmeur
- identifier des actions complexes qui deviendront peut-être des sous-programmes
- travailler à plusieurs en se partageant les actions complexes
- ...

Des outils de structuration

- Les sous-programmes : pour définir des bouts de codes réutilisables
- Les modules pour organiser les sous-programmes

Sommaire

Xavier Crégut <prenom.nom@enseeiht.fr>

Programme ou application

Définition

Un programme est une **suite finie** d'**opérations pré-déterminées** destinées à être **exécutées de manière automatique** par un **processeur** en vue d'effectuer des **traitements**, impliquant généralement une **interaction** avec son environnement.
Synonymes : script, application, logiciel.

Exemples de programmes

- Toutes les applications qui s'exécutent sur un ordinateur : traitement de texte, navigateur internet, messagerie instantanée, gestionnaire de fenêtres...
- Mais aussi sur console de jeu, imprimante, GPS, téléphone portable, décodeur TNT...
- Et : guichet automatique bancaire (GAB), injection électronique, ABS, pilote automatique...

Exemples d'environnements

- utilisateur humain : traitement de texte, calculatrice, etc.
- autre système informatique : GAB, navigateur Internet, etc.
- éléments physiques : capteurs et actionneurs (ABS, régulateur de vitesse...)

Programme dans un langage de haut niveau (Python)

decompte.py

```
1 # Objectif : Réaliser un décompte
2 # Auteur   : Xavier Crégut <Prenom.Nom@enseeiht.fr>
3 # Version  : 1.2
4
5 duree = 8            # durée du décompte  (> 0)
6 while duree > 0:     # décompte non terminé
7     print(duree)         #! afficher la durée restante
8     duree = duree - 1    #! diminuer la durée de 1
9 print('Go !')        #! Afficher le message « Go ! »
```

Exécution :

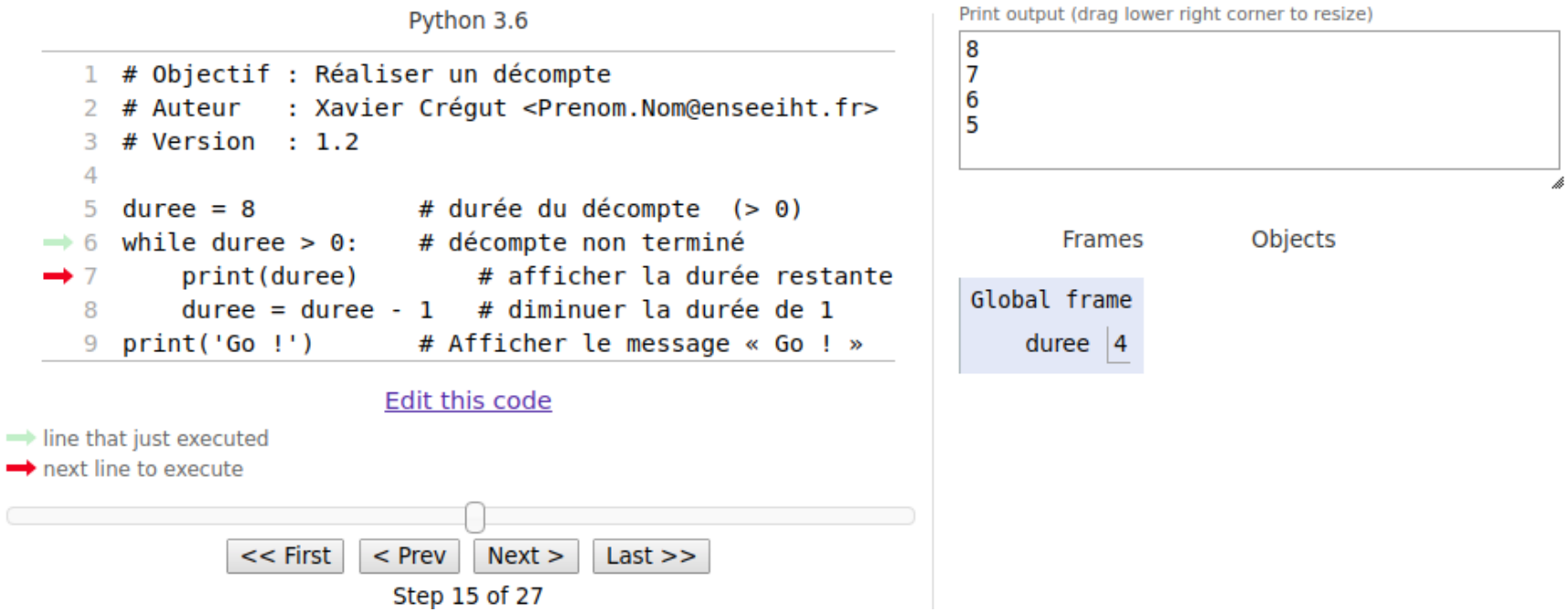
```
8
7
6
5
4
3
2
1
Go !
```

- duree est une variable qui correspond à un entier :
  - initialisée à 8, sa valeur est comparée à 0, affichée, décrémentée...
- des structures de contrôle (**while**, etc.) définissent l'ordre d'exécution des instructions
- **while** contrôle combien de fois on affiche et décrément la valeur de duree
- le décalage (indentation) des instructions montre qu'elles sont contrôlées par le **while**
- ce programme est éloigné de ce que c'est exécuter une machine...
- on ne sait ni comment, ni où sont stockées les données : à la charge de l'interpréteur Python
- interpréteur ?

Les commentaires de type #! sont des commentaires inutiles car ils paraphrases le code.

Exécuter un programme avec l'interpréteur en ligne Python tutor

- Python tutor est une application Web qui permet :
  - d'exécuter un programme instruction par instruction
  - de visualiser l'**évolution des valeurs des variables**, la mémoire, (cadre « Frames Objects »)
  - de voir ce qui est affiché dans le terminal (cadre « Print output »)
- Copier/coller le programme (attention à l'indentation), puis faire « Visualize Execution »
- Appuyer sur « *Next* » pour exécuter une instruction
  - la flèche rouge indique l'**instruction qui va être exécutée** (compteur ordinal)
  - la flèche vert clair indique l'instruction exécutée juste avant
  - sur la capture suivante, on vient d'exécuter 6 et la prochaine est 7
- Utiliser « *Next* » (et « *Prev* ») pour comprendre l'exécution de ce programme
  - « Edit this code » permet de revenir sur la page d'édition pour modifier le programme



Exécuter un programme sur un ordinateur

Constat

Un programme de haut niveau n'est pas directement exécutable par le processeur de l'ordinateur !

Interpréteur

Un interpréteur interprète un programme écrit dans un langage L1 pour l'exécuter directement.

Compilateur

Un compilateur traduit un programme écrit dans un langage L1 en un programme équivalent exprimé dans un langage L2.
Exemple : traduire le programme de haut niveau dans un langage que l'on sait exécuter (exemple : langage machine).

Remarque

Compilateurs et interpréteurs sont des programmes qui manipulent des programmes.

Analyse du programme

Constat

Dans les deux cas (compilation ou interprétation), l'outil doit comprendre le programme.

Principe

1. Le **texte source** est une suite de caractères
   - utiliser un éditeur de texte et non un traitement de texte ! (Notepad et non Word !)
2. Une phase d'**analyse lexicale** regroupe ces caractères en « mots » du langage
   - les commentaires sont ignorés (sur l'exemple caractère # jusqu'à la fin de la ligne)
   - les blancs ne sont pas significatifs et servent à séparer les mots (sauf ceux de l'indentation : ↪)
   - les « mots » sont : duree = 8 ⏎ while duree > 0 : ⏎ ↪ print (duree) ⏎ ↪ duree = duree - 1 ⏎ print ('Go !') ⏎
   - un « mot » est appelé « unité lexicale » ou « lexème » (« token » en anglais)
   - le lexème `while` est un mot-clé du langage Python
   - les lexèmes `8`, `1` ou `'Go !'` sont les constantes litérales (entier, chaîne de caractères)
   - duree est un nom (ici un nom de variable)
3. Une phase d'**analyse syntaxique** vérifie que les mots apparaîssent dans le bon ordre
   - ceci permet de donner un sens aux lexèmes
   - exemple : dans « `duree = duree - 1` », le « duree » de droite désigne la valeur de la variable nommée « duree » et celui de gauche désigne la variable « duree » à laquelle associer une nouvelle valeur.
4. Une phase d'**analyse sémantique** qui exploite ces données :
   - le compilateur fait des vérifications (existence des noms, typage...) et produit un « texte » dans un autre langage
   - l'interprêteur exécute le programme

Xavier Crégut <prenom.nom@enseeiht.fr>

Exemples d'erreurs

Python signale les erreurs lexicales comme des erreurs syntaxiques (`SyntaxError`)

Exemples d'erreurs lexicales (au chargement du programme)

```
x = !           # SyntaxError: invalid syntax
nom = 'Paul"    # SyntaxError: EOL while scanning string literal
2n              # SyntaxError: invalid syntax
...
```

Exemples d'erreurs syntaxiques (au chargement du programme)

```
d = 10          # ok
d < = 1         # SyntaxError: invalid syntax      (sur =)
while d > 0     # SyntaxError: invalid syntax      (il manque « : »)
2 = d           # SyntaxError: can't assign to literal
```

Exemples d'erreurs sémantiques (lors de l'exécution du programme)

```
d = 0           # ok
x = d + 10      # ok (x vaut 10)
y = d + 'cm'    # TypeError: unsupported operand type(s) for +: 'int' and 'str'
y = z           # NameError: name 'z' is not defined
y = 10 / d      # ZeroDivisionError: division by zero
```

Présentation des données

Exercice (1 minute)

Que représentent les lignes suivantes ? (répondre avec un ou plusieurs mots)

- A
- 10
- 12
- 1010
- X
- 𝍸 𝍸

Réponse

dix

Explications

On a présenté la quantité 10 de différentes manières :

- A : en hexadécimal (base 16) : 0 1 2 3 4 5 6 7 8 9 A B C D E F
- 10 : en base 10 (celle qu'on utilise usuellement : $1.10^1 + 0.10^0$)
- 12 : en base 8 ($1.8^1 + 2.8^0$)
- 1010 : en binaire ($1.2^3 + 0.2^2 + 1.2^1 + 0.2^0$)
- X : en chiffres romains
- 𝍸 𝍸 : en petits batons

Conséquences

- Il faudra choisir la manière de **modéliser (représenter, coder)** une donnée
  - Exemple : un mois représenté par un entier entre 1 et 12
- et la manière de la **présenter** (afficher, saisir) à l'utilisateur :
  - Exemple : un mois peut être affiché comme :
  - un entier de 1 à 12
  - une chaîne de caractères : `janvier`, `fevrier`...
  - une chaîne abrégée : `jan`, `fev`...
  - dans une autre langue : `jan`, `feb`...
  - ...
- La **modélisation** doit permettre d'effectuer efficacement les traitements sur les données
- La **présentation** doit être pratique, intuitive pour l'utilisateur

Synthèse

Nous avons vu dans ce chapitre :

1. Un exemple de programme en Python
2. Une brève évolution des langages de programmation
   - au début des langages très proches de la machine
   - maintenant des langages de haut niveau, plus proche du programmeur, voire du spécialiste d'un domaine métier (permettre à tous d'écrire ses programmes)
3. L'exécution d'un programme en Python avec Python tutor
4. Nous avons vu la structure des ces programmes :
   - l'aspect lexical : les mots utilisés
   - l'aspect syntaxique : les combinaisons de mots autorisées (les phrases)
   - l'aspect sémantique : donner du sens aux phrases
5. Ces programmes de haut niveau sont exploités par des outils pour être exécutés
   - soit un interpréteur : il exécute directement le programme
   - soit un compilateur : il traduit le programme d'un langage dans un autre
6. Enfin, nous avons terminé sur la différence entre représentation et présentation des données
   - on la représente (pour l'ordinateur) de manière à pouvoir faire efficacement des opérations dessus
   - on la présente (à l'utilisateur) pour qu'il la comprennent facilement et puisse interagir avec le programme

Sommaire

Objectifs

- Comprendre les instructions élémentaires de notre processeur (le langage Python)
- Se limiter au sous-ensemble « programmation impérative » de Python :
  - variables
  - expressions
  - instructions
  - structures de contrôle
- Connaître le minimum de la partie objet pour pouvoir utiliser les éléments de Python (str, etc.)
  - objet
  - méthode
- Écrire des programmes simples en Python
- Savoir utiliser des éléments fournis par des modules Python

Pourquoi Python ?

- **open-source**, compatible GPL et utilisations commerciales
- langage **multiplateformes**
- **bibliothèque** très riche et nombreux **modules** :
  - Cryptography, Database, Game Development, GIS (Geographic Information System), GUI, Audio / Music, ID3 Handling, Image Manipulation, Networking, Plotting, RDF Processing, Scientific, Standard Library Enhancements, Threading, Web Development, HTML Forms, HTML Parser, Workflow, XML Processing...
- importante **documentation** :
  - python.org : Tutorial, Language Reference, Library Reference, Setup and Usage
  - wiki.python.org
  - Python Enhancement Proposal (PEP)
  - The Python Package Index (PyPI)
  - stackoverflow
  - Notions de Python avancées sur Zeste de savoir
  - ...
- **outils de développement :**
  - IDE (*Integrated Development Environment*) : Idle, Spyder, Pycharm, etc.
  - Documentation : PyDOC, Sphynx, etc.
  - Tests : doctest, unittest, pytest, etc.
  - Analyse statique : pylint, pychecker, PyFlakes, mccabe, mypy, etc.
- des success stories :
  - Google, YouTube, Dropbox, Instagram,
  - Spotify, Mercurial, OpenStack, Miro, Reddit, Ubuntu...

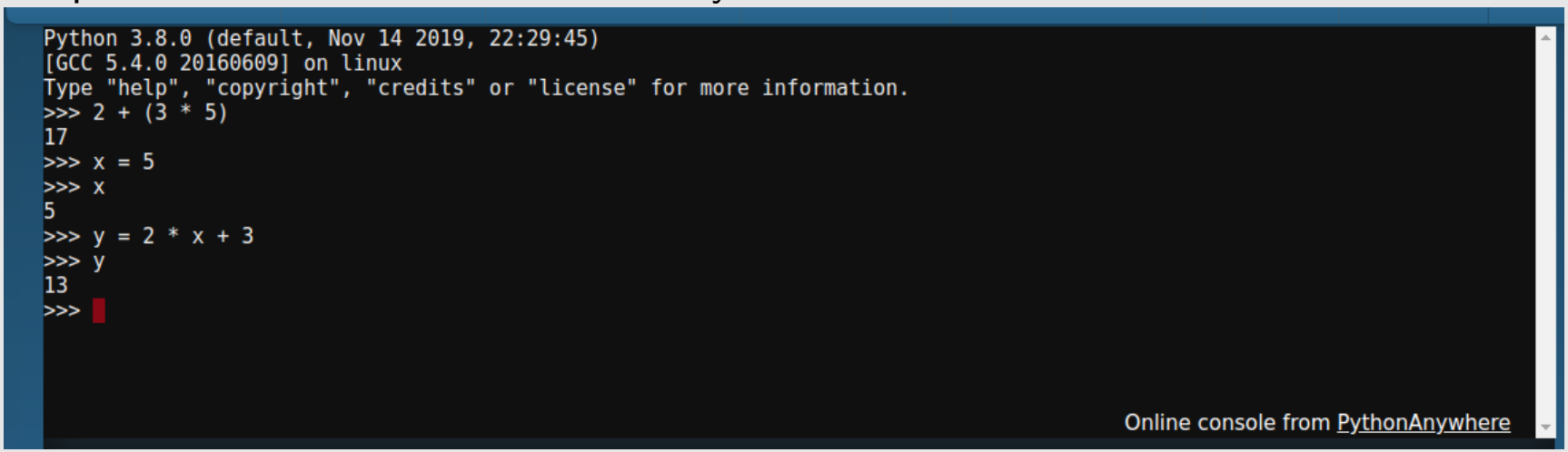
Console Python

Définition

Une console Python (*shell*) est un programme qui exécute les instructions Python au fur et à mesure

Intérêt : Pratique quand on n'a que quelques instructions simples à écrire

Python shell est une console Python (https://www.python.org/shell/).
On peut s'en servir de calculatrice. Essayez !



Meilleure solution

Pour un programme (plusieurs lignes), l'écrire dans un fichier avec l'extension .py.

Premier programme

Objectif : Afficher le périmètre d'un cercle dont l'utilisateur du programme saisit le rayon au clavier.

Exemple : L'utilisateur indique 5 pour le rayon

```
Rayon du cercle : 5
Le périmètre d'un cercle de rayon 5.0 est 31.41592653589
```

Le programme : Le lire et essayer de le comprendre (3 minutes)

perimetre_cercle.py

```
1  """Afficher le périmètre d'un cercle."""              #! Expliquer le contenu du fichier
2  import math                                           #! donner accès au module math
3
4  def afficher_perimetre_cercle():                      #! donner un nom à notre programme
5      """
6      Afficher le périmètre d'un cercle dont le rayon est demandé
7      au clavier à l'utilisateur.
8      """                                               #! docstring (sur plusieurs lignes)
9
10     # demander le rayon à l'utilisateur
11     saisie = input('Rayon du cercle : ')       #! une chaîne saisie au clavier
12     rayon = float(saisie)                      #! convertie en un nombre réel
13
14     # calculer le périmètre
15     perimetre = 2 * math.pi * rayon
16
17     # afficher le périmètre à l'utilisateur
18     print("Le périmètre d'un cercle de rayon", rayon, 'est', perimetre)
19
20 if __name__ == "__main__":          #! voir la section modules
21     afficher_perimetre_cercle()     #! exécuter le programme afficher_perimetre_cercle
```

Anatomie d'un programme

Les chaînes de caractères

- Entre apostrophes ou guillemets
- Le symbole peut être triplé : la chaîne peut alors s'étendre sur plusieurs lignes
- Pour **documenter le programme** (chaîne de début de fichier ou juste après `def` : docstring)
- Pour **dialoguer avec les utilisateurs** du programme (`'Rayon du cercle : '`, ...)

Le « programme » perimetre_cercle

- `def afficher_perimetre_cercle():` définir un programme (en fait un sous-programme).
- Ce n'est pas obligatoire mais constitue une **bonne pratique**
- Pour l'exécuter, il faut alors l'appeler en faisant `afficher_perimetre_cercle()`

Commentaires

- Les commentaires commencent par # et se terminent avec la fin de la ligne
- Ils permettent d'expliquer le programme
  - les commentaires avant un groupe de lignes sont issus des raffinages
  - les commentaires en fin de ligne expliquent l'instruction de cette ligne
- Ils sont ignorés lors de l'exécution du programme, ils sont utiles (nécessaires) pour la compréhension et la maintenance du programme par les humains
- Les `#!` sont des meta-commentaires : ne devraient pas être là, explications pour vous

Principaux constituants

Les variables (noms) pour manipuler les données

- Exemples : `saisie`, `rayon`, `perimetre`.
- Une variable permet d'accéder à une donnée (objet) :
  - `rayon` est initialisée avec le réel correspondant à la chaîne saisie par l'utilisateur
  - elle est utilisée pour calculer le périmètre et afficher le résultat <!–
- On peut préciser son type (`: str` et `: int`), ignoré par Python –>

IHM : Interface Humain-Machine

- IHM : gérer le dialogue avec l'utilisateur :
  - lui demander une donnée (`input`)
  - lui présenter des résultats (`print`)
- L'interface avec l'utilisateur est souvent une grosse partie du programme (ici les 3/4 !).
- Ce ne sera pas notre priorité dans ce module
- Nous nous limiterons à une IHM en mode texte en console

Autres éléments :

- Expression (un bout de phrase qui a une valeur) : `2 * math.pi * rayon`
- Instructions élémentaires (ou complexes) : `=`, `input`, `print`, `...`
- Structures de contrôle pour définir l'ordre d'exécution des instructions : `while`...
- Utiliser des choses définies dans d'autres modules : `import`
  - Ici, π est défini dans le module `math` et s'appelle `pi`

Xavier Crégut <prenom.nom@enseeiht.fr>

Exécutions de ce programme

Les exécutions suivantes peuvent être reproduites sur Python tutor.

Exécution nominale : l'utilisateur saisit un entier

```
> python afficher_perimetre_cercle.py
Rayon du cercle : 5
Le périmètre d'un cercle de rayon 5.0 est 31.41592653589793
```

Exécution nominale : l'utilisateur saisit un réel

```
> python afficher_perimetre_cercle.py
Rayon du cercle : 10.5e3
Le périmètre d'un cercle de rayon 10500.0 est 65973.44572538565
```

Exécution hors limite : l'utilisateur saisit un nombre négatif

```
> python afficher_perimetre_cercle.py
Rayon du cercle : -7.3
Le périmètre d'un cercle de rayon -7.3 est -45.867252742410976
```

Question : Ce dernier résultat est-il acceptable ?

Exécutions de ce programme (suite)

Exécution hors limite : l'utilisateur ne saisit pas un nombre

```
> python afficher_perimetre_cercle.py
Rayon du cercle : abc
Traceback (most recent call last):
  File "perimetre_cercle.py", line 21, in <module>
    afficher_perimetre_cercle()  #! exécuter le programme afficher_perimetre_cercle
  File "perimetre_cercle.py", line 13, in afficher_perimetre_cercle
    rayon: float = float(saisie) #! convertie en un nombre réel
ValueError: could not convert string to float: 'abc'
```

- Le programme se termine sur le message « Traceback (most recent call last): »
  - Le programme n'est pas **robuste**
- La dernière ligne indique l'**exception** qui s'est produite : `ValueError`
  - C'est donc un problème « d'erreur sur une valeur »
- ...avec l'**explication** : `could not convert string to float: 'abc'`
  - En effet 'abc' ne correspond pas à un nombre réel !
- La trace des appels est affichée et vous permet de localiser l'erreur
  - le programme a été exécuté jusqu'à la ligne 21 où `afficher_perimetre_cercle` est appelée
  - le programme `afficher_perimetre_cercle` s'exécute jusqu'à la ligne 13 où l'exception se produit
  - la conversion de la chaîne saisie par l'utilisateur (`abc`) en réel provoque l'exception

Attention : Il est important de comprendre une telle trace des appels pour savoir où intervenir dans le programme quand il se termine sur une exception.

Objet

Toutes les données manipulées par Python sont des objets : **tout est objet**.

```
421              # un entier (int, integer)
3.9              # un nombre réel (float)
4.5e-10          # un autre avec un exposant
3+5j             # un nombre complexe (complex)
"Bonjour"        # une chaîne de caractères (str, string)
'x'              # aussi une chaîne de caractères !
[1, 2.5, 'a' ]   # une liste (list)
None             # objet particulier qui signifie rien.
```

Un objet possède :

- une **identité** : entier unique et constant durant toute la vie de l'objet.
  On l'obtient avec la fonction `id` (l'adresse en mémoire avec CPython)

  ```
  id(421)      # 140067608358512 (par exemple !)
  ```

- un **type** (la classe à laquelle il appartient) que l'on obtient avec la fonction `type`
  Le type d'un objet ne peut pas changer.

  ```
  type(421)    # <class 'int'>
  type(3.9)    # <class 'float'>
  type('x')    # <class 'str'>
  ```

- des **opérations** généralement définies au niveau de son type

Type

Un **type** définit les caractéristiques communes à un ensemble d'objets.

- Parmi ces caractéristiques, on trouve les **opérations** qui permettront de manipuler les objets.
- La fonction `dir` fournit tous les noms définis sur un type.
- La fonction `help` permet d'obtenir la documentation d'un objet.

```
>>> dir(str)        #! affiche les noms qui sont définis sur `str`
[..., '__doc__', ..., 'capitalize', 'count', 'endswith', 'format', 'index',
'isalnum', 'isalpha', 'islower', 'isnumeric', 'join', 'lower', 'replace',
'split', 'startswith', 'strip', 'swapcase', ...]

>>> help('str.lower')   #! donne la description de la méthode `lower` de `str`.
str.lower = lower(...)
    S.lower() -> str           #! Appliquer sur S, lower() fournit une chaîne (str)

    Return a copy of the string S converted to lowercase.    #! explications
```

Essayer dans Python shell :

```
help(str)        #! affiche la documentation de 'str'
```

Opération

Parmi les **opérations**, on peut distinguer :

- les **opérateurs** « usuels » : + - * / ** ...
- les **sous-programmes** (procédures[1] ou fonctions) : print, len, id, type, etc.
- les **méthodes** (sous-programmes définis sur les objets) : lower, islower, etc.

Chaque type d'opération a sa **syntaxe d'appel** :

- opérateurs : souvent infixe : `1 + 2 * 4` (comme en math)
- sous-programme : `print(2, 'm')` ou `len('Bonjour')`
- méthode : notation pointée : `'Bonjour'.lower()`

```
2 * 4                   #! 8
2 ** 4                  #! 16    (puissance)
print(2 ** 4)           #! 16    (une procédure)
print(2, 'm')           #! 2 m
len('Bonjour')          #! 7 : nombre de caractères de la chaîne (une fonction)
'Bonjour'.lower()       #! 'bonjour'              (une méthode)
'Bonjour'.islower()     #! False                  (une autre méthode)
```

1. En Python, tout est fonction !

Nom (ou Variable)

- Une **variable** permet de référencer un objet grâce à un **nom** (**identifiant**).
  - En Python, on parle plutôt de **nom** que de **variable**.
  - Un nom (une variable) est en fait un **accès**, une **référence**, un **pointeur** sur un objet.
  - **Règle :** Un nom est de la forme : lettre ou souligné, suivi de chiffres, lettres ou soulignés
- **Intérêt :** nommer les objets pour y accéder (et améliorer la lisibilité du code).
  $\Longrightarrow$ Toujours choisir un nom **significatif** !

```
longueur_cm = 10        #! le nom `longueur_cm` est associé à l'entier 5
UN_POUCE = 2.54         # centimètres
longueur_pouce =  UN_POUCE * longueur_cm          #! utilise  les noms précédents
print(longueur_cm, "cm =", longueur_pouce, "p")    #! ici aussi
```

- **Convention :**
  - Un nom de variable est en minuscules.
  - Utiliser un souligné _ pour mettre en évidence les morceaux d'un nom : `prix_ttc`
  - Un nom en majuscules désigne une constante (on ne devrait pas l'associer à un autre objet)
  - Voir PEP 8 – Style Guide for Python Code
- **Exemples :**
  - Corrects : rayon, perimetre_cercle, prix_ttc, prix_ht, n1, n2.
  - Déconseillés : prixttc, lageducapitaine, rayon_du_cercle (trop long), r (trop court)
  - Incorrects : 2n
- **del :** instruction qui permet de supprimer un nom (`del longueur_cm`)

Quelques types prédéfinis

entier (int)

- entiers relatifs (positifs ou négatifs)
- exemples : `-153`, `0`, `2048`
- arbitrairement grands

```
2 ** 200 # 1606938044258990275541962092341162602522202993782792835301376
```

réel (float)

- nombres à virgule flottante
- exemples : `4.5` `5e128` `-4e-2` `1.12E+30`
- exposant de -308 à +308, précision 12 décimales, Voir IEEE 754

booléen (bool)

- deux valeurs possibles : `False` et `True`
- une variable booléenne peut être initialisée avec une expression booléenne :
  - `est_majeur = age >= 18`
  - `est_chiffre = len(chaine) == 1 and '0' <= chaine <= '9'`
  - ‘incorrect = not (1 <= mois <= 12)

Opérateurs

opérateurs arithmétiques

opération	résultat	exemple avec x = 10; y = 3
x + y	somme	13
x * y	produit	30
x - y	soustraction	7
x / y	division réelle	3.3333333333333335
x // y	division entière	3
x % y	reste de la division entière	1
- x	opposé	-10
+ x	neutre	10
x ** y	puissance	1000
abs(x)	valeur absolue	abs(-10) donne 10
int(x)	conversion vers un entier	int(3.5) donne 3, comme int('3')
float(x)	conversion vers un réel	float(10) donne 10.0, comme float('10')

Remarque : Mettre en espace avant et après les opérateurs binaires (lisibilité).

Opérateurs relationnels

`<  <=  >  >=  ==  !=`

- Ce sont les opérateurs usuels. Notation spécifique pour l'égalité (==) et la différence (!=)
- Forme contractée : `1 <= mois <= 12` équivalente à `1 <= mois and mois <= 12`

Opérateurs logiques

`and or not`

Ce sont les opérateurs de la logique que l'on utilise tous les jours !

évaluation en court-circuit : `(n != 0) and (s / n >= 10)`

- si `n` vaut 0, `n != 0` est faux, et donc le résultat est faux sans évaluer s/n (division par zéro !)

Formules de De Morgan (la logique usuelle, de tous les jours)

```
not (a and b) <==> (not a) or  (not b)
not (a or b)  <==> (not a) and (not b)
not (not a)   <==> a
```

Remarque : Tout objet peut être considéré comme booléen. **Ne pas en abuser.**
Voir valeurs booléennes

Priorité des opérateurs

Inutile de lire en détail. On s'en servira sur les planches suivantes.

P = priorité. Plus le numéro est petit, plus la priorité est faible.

P	Operator	Description
1	`lambda`	Lambda expression
2	`if – else`	Conditional expression
3	`or`	Boolean OR
4	`and`	Boolean AND
5	`not x`	Boolean NOT
6	`in`, `not in`, `is`, `is not`, `<`, `<=`, `>`, `>=`, `!=`, `==`	Comparisons, including membership tests
7	`|`	Bitwise OR
8	`^`	Bitwise XOR
9	`&`	Bitwise AND
10	`<<`, `>>`	Shifts
11	`+`, `-`	Addition and subtraction
12	`*`, `@`, `/`, `//`, `%`	Multiplication, division, remainder
13	`+x`, `-x`, `~x`	Positive, negative, bitwise NOT
14	`**`	Exponentiation
15	`await x`	Await expression
16	`x[index]`, `x[index:index]`, `x(arguments...)`, `x.attribute`	Subscription, slicing, call, attribute reference
17	`(expressions...)`, `[expressions...]`, `{key: value...}`, `{expressions...}`	Binding or tuple display, list display, dictionary display, set display

L'associativité est à gauche, sauf pour ** ou les opérateurs unaires (associativité à droite).

Comment s'évalue l'expression suivante ?

```
x != y * 3 - z + 2
```

Principe

1. Identifier les mots (–) et trouver la priorité des opérateurs grâce à la table précédente

```
x  !=  y  *  3  -  z  +  2
-  --  -  -  -  -  -  -  -   # les mots (les -- marquent les mots)
-  6   -  12 -  11 -  11 -   # les mots « opérateurs » remplacés par leur priorité
```

2. Associer à l'opérateur de priorité la plus forte les expressions à gauche et à droite
   - ici c'est * (priorité 12)
   - ses opérandes sont y à gauche et 3 à droite
   - on met des parenthèses autour pour marquer l'association de l'opérateur et ses opérandes

```
x  !=  (y  *  3)  -  z  +  2
-  6   ---------  11 -  11 -
```

3. Continuer ainsi.
   - - et + ont même priorité (11), on prend le plus à gauche car ils sont associatifs à gauche
   - on considère donc - qui a comme opérande gauche (y * 3) et z à droite

```
x  !=  ((y  *  3)  -  z)  +  2
-  6   -----------------  11 -
```

4. On continue avec + (priorité 11)

```
x  !=  (((y  *  3)  -  z)  +  2)
-  6   -----------------------
```

5. Et enfin l'opérateur !=

```
(x  !=  (((y  *  3)  -  z)  +  2))
```

Xavier Crégut <prenom.nom@enseeiht.fr>

Exercice : Expressions, opérateurs et priorité

1. Parenthéser les expressions suivantes (pour expliciter l'ordre de calcul)

```
1 x != y * 3 + 5
2 n != 0 and s / n >= 10
3 y >= b and z ** 2 != 25 and not x in e
```

2. Comment s'évaluent les expressions suivantes ?

```
1 x = 12
2 y = 6 - 3 - 2
3 z = 2 ** 1 ** 3
```

3. Que signifient les expressions suivantes ?

```
1 0 <= x <= 9
2 x < 10 < y
3 x < 10 >= y
4 x < 10 == y
5 x == y == 0
```

4. Réécrire les expressions suivantes sans utiliser `not`

```
1 not (1 <= mois and mois <= 12)
2 not (reponse == 'oui' or reponse == 'non')
```

Conseil : Éviter les expressions compliquées : utiliser des noms pour les simplifier et les expliquer !

Exercices à vérifier ou faire sur Python shell.

Exercice : Opérateurs artithmétiques

Pour chacune des questions, répondre avant de vérifier sur l'interpréteur Python si votre réponse est juste.

1. Quel est le résultat de 9 / 4 ?
2. Quel est le résultat de 9 // 4 ?
3. Quel est le résultat de 9 % 4 ?
4. Comment obtenir 2^{10} et quelle est sa valeur ?
5. Quels sont la valeur et le type de 8 / 4 ?

Exercice : Chiffres d'un entier

Soit un entier n, quelle expression permet d'obtenir :

1. le chiffre des unités (1 pour 421, 5 pour 35, 0 pour 0...)
2. le chiffre des dizaines (2 pour 421, 3 pour 35, 0 pour 0...)
3. le chiffre des dizaines **et** le programme ne doit utiliser que 10 comme constante littérale
4. le chiffre des centaines

Instructions

Nous allons voir les principales instructions :

- l'affectation : associer une nouvelle valeur à une variable
- `print` : écrire sur le terminal
- `input` : demander une valeur à l'utilisateur (via le terminal)
- `pass` : instruction qui ne fait rien
- `assert` : vérifier qu'une expression est vraie
- structures de contrôle : contrôler l'ordre d'exécution des instructions précédentes

- Attention, il est important de comprendre l'effet de chaque instruction (et structure de contrôle).
- L'effet est décrit dans la partie « exécution » dans les pages suivantes.
- Ne pas comprendre l'effet d'une instruction, c'est ne pas comprendre un programme et ne pas être capable d'en écrire un !

Affectation

Syntaxe : `nom = expression`

Définition : L'affectation associe la valeur d'une expression (un objet) à un nom.

Exécution :

1. Évaluer l'expression (à droite de =)
2. Associer cette valeur au nom (à gauche de =)

- Remarque : si le nom n'existe pas déjà, le nom `nom` est créé

```
prix_ht = 83.25
description = 'Un super produit'
tva = .20
prix_ttc = prix_ht * (1 + tva)    # 99.90 (ou presque : 99.89999999999999)
prix_ht = 100.0
prix_ttc                          # toujours presque 99.90
```

Initialiser plusieurs noms avec le même objet

```
a = b = c = 0        # a, b et c associés à l'objet 0
```

Formes contractée

- **Principe :** x #= y est équivalent à x = x # y (# étant un opérateur binaire)
- **Exemple :** x += 2 est équivalent à x = x + 2

Exercice à faire sur Python tutor ou Python shell

Exercice : Calculer x^5

Soit x un nombre réel, calculer x^5 (on l'appellera x5) :

1. en utilisant l'opérateur puissance de Python
2. en n'utilisant que l'opérateur multiplication
3. en n'utilisant que l'opérateur multiplication et en faisant au plus 3 multiplications

Affectation multiple

Syntaxe :

```
nom1, nom2, ..., nomN = expression1, expression2, ..., expressionN
```

Contrainte : Autant de noms à gauche que d'expressions à droite

Exécution :

1. Évaluer toutes les expressions (à droite)
2. Associer le i^{e} nom (pour i de 1 à N, i.e. de gauche à droite) avec la valeur de la i^{e} expression

- **Remarque :** Toutes les expressions sont évaluées avant que les affectations commencent !

```
nom, age = 'Paul', 18     # équivalent à : nom = 'Paul'; age = 18    Utile ?
a, b, c = 1, 2 ** 3, -1   # a == 1 and b == 8 and c == -1          Lisible ?
a, b = b, a               # a == 8 and b == 1                        Utile !
```

Remarque : On reverra cette notation avec les séquences...

Typage en Python

Quel est le type d'un nom ? Quel est le type de `x` dans le programme suivant ?

```
x = 5
print(x)
x = "non"; print(x)   #! ; car plusieurs instructions sur la même ligne (déconseillé)
x = 2.17;  print(x)
x = 2;     print(x)
```

Un nom n'a pas de type. Il a le type de l'objet qui lui est associé. On parle de **typage dynamique**.

On peut préciser les informations de type en Python grâce aux décorateurs (après « : »).

```
x: int = 5    #! x déclaré du type int
print(x)
x = "non"; print(x)
x = 2.17;  print(x)
x = 2;     print(x)
```

```
> python exemple_typage_solution.py
5
non
2.17
2
```

Mais ces informations sont **ignorées par l'interpréteur Python** (voir exécution à droite).

Des programmes peuvent les exploiter comme mypy, un **vérificateur de type pour Python**

```
> mypy exemple_typage_solution.py
exemple_typage_solution.py:3: error: Incompatible types in assignment (expression has type "str"
exemple_typage_solution.py:4: error: Incompatible types in assignment (expression has type "floa
Found 2 errors in 1 file (checked 1 source file)
```

mypy fait de l'inférence de type. Donc mêmes résultats avec le premier programme.

Typage dynamique

Python s'appuie exclusivement sur du **typage dyamique**.

Conséquence : Seules les erreurs de syntaxe sont détectées lors du chargement d'un programme par l'interpréteur Python. Les autres erreurs sont signalées à l'exécution. **Il faut donc tester !**

Conseil : Pour détecter avant l'exécution des erreurs de type dans les programmes :

1. Définir les types des variables
2. Utiliser un outil tel que mypy

Tester le type d'un objet à l'exécution

Pour savoir si un objet est d'un certain type, on peut utiliser isinstance :

```
x = 421
isinstance(x, int)    # True          isinstance(x, float)  # False
isinstance(x, str)    # False         isinstance(x, bool)   # False
```

Instruction de sortie : **print**

print permet d'écrire un ou plusieurs objets :

- en les séparant par un espace (`sep=' '` par défaut)
- en ajoutant un retour à la ligne à la fin (`end='\n'` par défaut)

```
a, b = 18, 'ok'
print('a =', a, 'et b =', b)     # a = 18 et b = ok
print('a =', a, 'et b =', b, sep='__', end='~!\n')
    # a =__18__et b =__ok~!
```

La méthode **str.format** permet de simplifier certaines écritures

```
print('a = {} et b = {}'.format(a, b))  # a = 18 et b = ok
    # les {} correspondent aux paramètres de format (dans l'ordre)
print('{0}, {1} et {0}'.format(a, b))   # 18, ok et 18
    # le numéro entre {} est le numéro du paramètre à écrire
print('{:7.2f}'.format(1.2345))      #    1.23 (avec des espaces devant)
    # format après « : » : 7 caractères dont 2 pour la partie décimale
```

- les **chaînes formatées** (*f-strings*) depuis Python 3.6 :
- on ajoute un f devant la chaîne
- on peut mettre une expression Python entre accolades, sa valeur sera écrite

```
print(f'{a}, {b} et {a // 4:3}')                          # 18, ok et   4
nombre, demi_largeur, decimales = 1.2345, 5, 3
print(f"nombre = {nombre:{2 * demi_largeur}.{decimales}f}")    # nombre =      1.234
```

Il existe d'autres façons d'écrire en Python !

Xavier Crégut <prenom.nom@enseeiht.fr>

Exercice : Afficher le résultat d'opérations

Soit a = 503 et b = 17, afficher, en utilisant a et b et aucune constante littérale, le texte suivant « 503 / 17 = 29.58823529411765 » en utilisant `print` :

1. avec plusieurs paramètres (séparés par des virgules).
2. avec la méthode `format`.
3. en utilisant les chaînes formatées

Exercice : Améliorer l'affichage des nombres réels

Reprendre l'exercice précédent en affichant le résultat de la division avec 3 chiffres après la virgule.

Exercice : afficher plusieurs objets

On considère l'instruction `print(1, 2, 3, 4, 5, 6)`.

1. Quel est l'effet de l'instruction précédente ?
2. Compléter l'instruction après le paramètre 6 pour obtenir l'affichage suivant :

```
1 -> 2 -> 3 -> 4 -> 5 -> 6...
```

Exercice : Réaliser un affichage tabulé

Soit les variables suivantes : `nom = 'Paul' ; note = 15.5 ; matiere = 'programmation'`.
Écrire dans l'ordre le nom sur 7 positions, la note sur 6 positions avec 2 chiffre après la virgule et la matière sur 10 positions (« `Paul  15.50 programmation` ») en utilisant :

1. la méthode `format` de `str`
2. les chaînes formatées

Instruction de saisie : **input**

`input(prompt = '') -> str` : demande à l'utilisateur du programme une information

- prompt est le texte qui sera affiché à l'utilisateur (chaîne dans les parenthèses)
- la réponse de l'utilisateur est une chaîne de caractères (ceux tapés au clavier)

```
>>> reponse = input('Votre choix ? ')
Votre choix ? quitter
>>> reponse
'quitter'
```

Si on attend un entier ou réel, il faudra **convertir la chaîne** obtenue

```
>>> reponse = input('Un entier : ')
Un entier : 15
>>> reponse
'15'
>>> n = int(reponse)    # conversion d'une chaîne en entier
>>> n
15
>>> n = float(input())  # Attention, il faudrait mettre un message pour l'utilisateur
>>> n
4.5
```

Attention : Exception `ValueError` si la chaîne ne correspond pas à un entier (resp. un réel).

Exercice à faire sur Python tutor.

Exercice : Saisie au clavier

On veut avoir le dialogue suivant avec l'utilisateur. Après les points d'interrogation apparaît ce qui a été saisi par l'utilisateur.

```
Nom ? Paul
Note ? 15.5
Coefficient ? 4
```

1. Réaliser cette saisie sachant que l'on veut initialiser les variables :
   - `nom` : une chaîne de caractère,
   - `note` : un réel et
   - `coefficient` : un entier.
2. Que se passe-t-il si l'utilisateur répond « dix » pour la note ?
3. Que se passe-t-il si l'utilisateur répond « 4.5 » pour le coefficient ?

Instruction **pass**

Définition

L'instruction **pass** est une instruction qui ne fait rien.

Intérêt

Elle est utile quand une instruction est attendue syntaxiquement mais qu'on n'a rien à mettre (au moins pour l'instant).

Exemple

```
def mon_programme():
    pass    #! Le vrai code n'est pas encore écrit
```

Il faut au moins une instruction après `def` !

Donner la chaîne de documentation serait suffisant mais `pass` montre que le vrai code manque.

Instruction **assert**

```
assert condition
assert condition, message        # avec un message d'explication
```

1. évalue la condition
2. si elle est vraie, ne fait rien, sinon « arrête » le programme sur l'exception AssertionError

```
# Saisir deux entiers a et b > 0
...
...

assert a > 0
assert b > 0, 'b == ' + str(b)
```

```
a = -5
b = -7
Traceback (...):
  File "saisir_a_b.py", line 5
    assert a > 0
AssertionError
```

```
a = 4
b = -3
Traceback (...):
  File "saisir_a_b.py", line 6
    assert b > 0, 'b == ' + str
AssertionError: b == -3
```

Intérêt : Moyen simple de vérifier que les hypothèses faites lors de la conception et l'implantation du programme sont effectivement respectées lors de son exécution.

- On peut désactiver l'évaluation des **assert** en lançant l'interpréteur Python avec l'option `-O`
- **Conséquence :** Ne jamais écrire un programme dont l'exécution exploite l'effet de **assert**

Structures de contrôle

Objectif

Les **structures de contrôle** décrivent l'ordre dans lequel les instructions seront exécutées.

- enchaînement séquentiel (bloc ou séquence),
- traitements conditionnels (**if** ... **elif** ... **else**)
- traitements répétitifs (**while** et **for**)
- mécanisme d'exception (plus tard)
- appel d'un sous-programme (bientôt)

Bloc

Bloc

Un **bloc** est une suite d'instructions qui sont exécutées dans l'ordre de leur apparition.
Synonymes : séquence ou **instruction composée**

Règle

- Toutes les instructions d'un méme bloc doivent avoir exactement la même indentation.
- La ligne qui précède le bloc se termine par un deux-points « : »

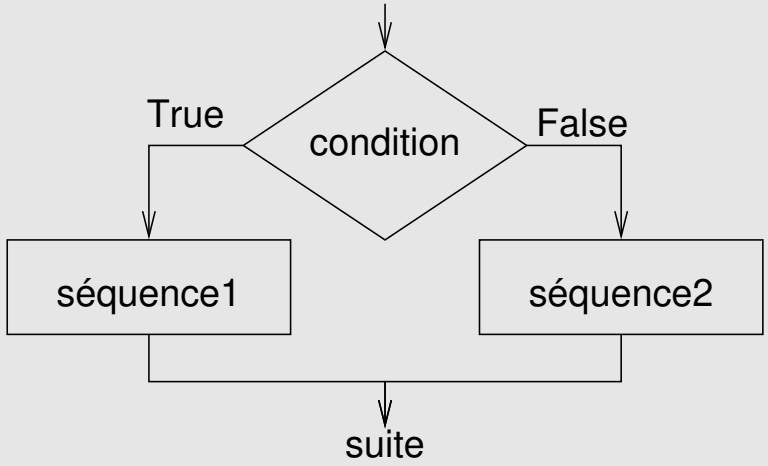
Conseil

Ne pas mélanger espace et tabulation dans les indentations.
Python recommande de n'utiliser que les espaces et une indendation de 4 espaces.

conditionnelle **if** (Si)

Conditionnelle if . . . else . . .

```
if condition:
    blocSi
else:
    blocElse
suite
```



Exécution :

- La condition est évaluée.
- Si elle est vraie alors *blocSi* est exécuté puis `suite`
- Si elle est fausse alors *blocElse* est exécuté puis `suite`

Propriétés

- Les deux blocs sont exclusifs
- La partie **else** est optionnelle
- Dans le `else`, la condition est fausse ; on a « `not condition` »

Exemple : Signe d'un entier

```
if n1 == n2:
    resultat = 'égaux'
else:   # n1 != n2
    resultat = 'différents'
```

Exemple sans else

```
if n > max:
    max = n
```

Exercice : Attention à l'indentation

vérifier avec Python tutor

1. Est-ce que les deux programmes suivants sont équivalents (pour toute valeur de a) ?
2. Les exécuter avec Python tutor pour confirmer (essayer avec a = 5 et avec a = 2).

indentation1.py

```
a = 5 # ou a = 2
if a > 4:
    a += 1
    print(a)
```

indentation2.py

```
a = 5 # ou a = 2
if a > 4:
    a += 1
print(a)
```

Exercice : majeure ou mineure

majorite.py

Étant l'age d'une personne, écrire « majeure » si elle est majeure et « mineure » sinon.

Partie **elif** (SinonSi)

SinonSi : elif

- On peut ajouter après le **if** des **elif** (SinonSi) avec la sytaxe suivante :

```
if condition1:
    assert condition1
    bloc1
elif condition2:
    assert not condition1 and condition2
    bloc2
...
elif conditionN:
    assert not condition1 and ... and not conditionN-1 and conditionN
    blocN
else:
    assert not condition1 and ... and not conditionN
    blocE
suite
```

Exécution

- Les conditons sont évaluées dans l'ordre
- Dès qu'une condition est vraie, le bloc associé est exécuté, puis l'exécution continue à 'suite'
- Si aucune condition n'est vraie, le *blocE* du **else** est exécuté puis l'exécution continue à 'suite'

Exemple : signe d'un entier

```
if n > 0:
    resultat = 'positif'
elif n < 0:      # not (n > 0) and (n < 0)
    resultat = 'negatif'
else:            # not (n > 0) and not (n < 0)
    resultat = 'nul'
```

Exécuter ce programme

Voir le transparent suivant pour voir comment exécuter ce programme.

Exercice

Réécrire la conditionnelle précédente sans utiliser `elif`.

Quelle est la version la plus lisible ?

- `elif` est équivalent à «`... else: if ...`» en évitant un niveau d'indentation supplémentaire
- **Intérêt :** les différents cas sont clairement exclusifs : même indentation pour chaque cas

Exécuter un programme

- Dans ce chapitre, on ne traite pas le dialogue avec l'utilisateur du programme
- Par exemple, pour le signe d'un entier :
  - on suppose que `n` est déjà initialisé
  - on n'affiche pas le résultat
- Cependant, pour exécuter ce programme, il faudra donner une valeur à `n`
- Le plus simple est de définir `n` en début de programme

```
n: int = 5
```

- On peut alors exécuter le programme.
- Avec Python tutor, on n'a pas besoin de plus puisque la valeur des variables est affichée
- Avec les autres outils, il faudra ajouter un affichage à la fin

```
print(resultat)
```

- Pour tester avec d'autres valeurs de n, on peut ajouter de nouvelles affectations
  - ainsi on se souvient de toutes les valeurs de `n` essayées
  - on peut facilement permuter les affectations pour refaire une ancienne exécution

```
n: int
n = 5
n = -4
n = 0
```

- On peut aussi remplacer l'affectation par une saisie (ne pas oublier la conversion)

```
n: int = int(input())
```

- À essayer !

Xavier Crégut <prenom.nom@enseeiht.fr>

Exercice : Tarif d'une place

tarif_place.py

Le tarif normal de la place est de 12,60 €. Les enfants (moins de 14 ans) paient 7 €. Les séniors (65 ans et plus) paient 10,30 €. Étant donné son age, combien une personne doit-elle payer ?

Exemples :

- Pour un age de 25 ans, le tarif est 12.60 €.
- Pour un age de 8 ans, le tarif est 7 €.
- Pour un age de 75 ans, le tarif est 10.30 €.
- Pour un age de 14 ans, le tarif est 12.60 €.

Exercice : Nombre de jours d'un mois

nb_jours_mois.py

Étant donné un numéro de mois compris entre 1 et 12, déterminer son nombre de jours dans le cas où l'année n'est pas bissextile.

Exemples :

- Le mois 1 a 31 jours
- Le mois 2 a 28 jours
- ...
- Le mois 12 a 31 jours

Exercice : La plus grande de trois valeurs

max3.py

Afficher la plus grande de trois variables entières a, b et c.
Contrainte : On n'utilisera pas d'autres variables et seulement des conditions simples (un opérateur et deux opérandes), le moins possible.

Exemples :

```
a   b   c  ->  max
5   9   1  ->  9
3   1   8  ->  8
4   4   4  ->  4
```

Formes équivalentes au **if**

Formulations à éviter !

```
if condition:                    |  if condition:
    resultat = True              |      resultat = False
else:                            |  else:
    resultat = False             |      resultat = True
```

Les formulations précédentes, à éviter, peuvent être réécrites avec une simple expression booléenne :

```
resultat = condition             |  resultat = not condition
```

Exemple : Il est *majeur* ssi son *age* est supérieur ou égal à 18 : `est_majeur = age >= 18`

Si Arithmétique

```
if condition:
    resultat = valeurVrai
else:
    resultat = valeurFaux
```

peut se réécrire en :

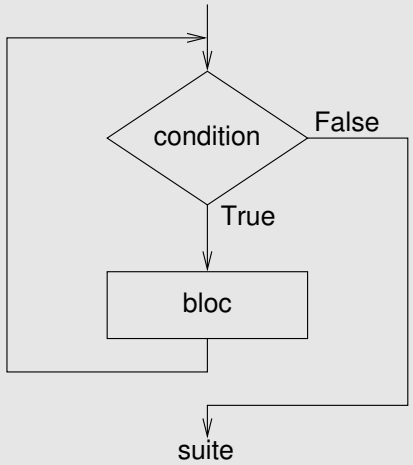
```
resultat = valeurVrai if condition else valeurFaux
```

Intérêt : Il s'agit d'une expression et non d'une instruction
Critique : Lisible ?

Répétition **while** (TantQue)

Syntaxe :

```
while condition:
    bloc
suite
```



Exécution :

1. Évaluer la condition
2. Si elle est vraie le bloc est exécutée et on recommence en 1
3. Si elle est fausse l'exécution continue à `suite`

Exemple : somme des n premiers entiers

somme_cours.py

```
1 n = ...
2 somme: int = 0       # la somme des entiers
3 i: int = 1           # pour parcourir les entiers de 1 à n
4 while i <= n:  # i est à prendre en compte
5     somme = somme + i     #! ou somme += i
6     i = i + 1             #! ou i += 1
7 print(f'la somme des entiers de 1 à {n} est : {somme}')
```

Exercice : Exécuter à la main ce programme avec n = 3, puis avec Python tutor
Question : Peut-on faire ce calcul plus efficacement ?

Constat : On peut ne pas exécuter *bloc* (*condition* fausse au départ)

Xavier Crégut <prenom.nom@enseeiht.fr>

Terminaison des boucles : une condition nécessaire

Avec les boucles, les problèmes commencent !

Avec une boucle, les mêmes instructions peuvent être exécutées plusieurs fois : on revient en arrière dans le programme.
Comment alors être sûr qu'il va s'arrêter ?

```
i = 1
while i < 10:
    print(i)
```

Est-ce que ce programme s'arrête ? Pourquoi ? Le corriger.

Conséquence

Règle : *bloc* doit modifier *condition* (sinon boucle infinie ou code mort)

Dans l'exemple précédent, on ne modifie pas `i` : on peut ajouter `i += 1` dans le `while`.

Attention : c'est une **condition nécessaire, pas suffisante !**

Autre exemple

```
i = 1
while i < 10:
    print(i)
    i -= 1
```

1. Est-ce que le bloc modifie la condition ?
2. Est-ce que le programme se termine ?
3. Comment être sûr qu'un programme se termine ?

Xavier Crégut <prenom.nom@enseeiht.fr>

Terminaison des boucles : une condition suffisante

Variant (définition)

Quantité entière positive qui décroit strictement à chaque passage dans la boucle.

Terminaison d'une boucle

Pour démontrer qu'une répétition (boucle) se termine il faut :

1. identifier son variant
2. et démontrer ses propriétés : entier, positif, décroissance stricte.

Exemple

Variant de "somme des n premiers entiers" :

- formel (une expression Python ou mathématique) : `n - i + 1`
- informel (en langage naturel) : **nombre d'entiers restant à sommer**

Exercice (2 minutes)

Se convaincre que « `n - i + 1` » est bien un variant de la somme des n premiers entiers

Correction d'une répétition

Problème

La boucle calcule-t-elle ce qu'on souhaite ? Par exemple, la somme des n premiers entiers ?

Invariant (définition)

Propriété toujours vraie, avant et après chaque passage dans la boucle

Exemple : Invariant de "somme des i premiers entiers"

- formel : **somme ==** $\sum_{k=0}^{i-1} k$
- informel : ***somme* est la somme des entiers de 0 à i-1**

Exercice (4 minutes)

1. Se convaincre que somme == $\sum_{k=0}^{i-1} k$ est bien un invariant.
2. A-t-on toutes les informations pour montrer la correction ?

Conseil :

À défaut de formaliser l'invariant, le donner en langage naturel (commentaire)

Xavier Crégut <prenom.nom@enseeiht.fr>

Correction d'une répétition (fin)

Correction d'une répétition : invariant (et variant et condition de sortie)

Les trois propriétés suivantes vraies à la sortie de la boucle doivent montrer la correction :

1. L'invariant est vrai : I
2. Le variant est positif ou nul : $V \geq 0$
3. La condition du `while` est fausse : `not condition`

Exercice (3 minutes)

Montrer que la boucle qui calcule la somme des n premiers entiers est correcte

Formulation des variants et invariants

Forme générale

```
...
while condition:
    # Variant : V
    # Invariant : I
    bloc
#! Après le while on a :  not condition and V >= 0 and I
```

Application à somme des n premiers entiers

somme_cours_invariant.py

```
n: int = ...
assert n > 0    #! hypothèse sur n
somme: int = 0
i: int = 1
while i <= n:
    # Variant : n - i + 1
    #     nombres d'entier restant à sommer
    # Invariant : somme =  0 + 1 + ... + i - 1
    #     somme est la somme des entiers de 0 à i-1
    somme = somme + i
    i = i + 1
```

Xavier Crégut <prenom.nom@enseeiht.fr>

Démonstrations variant, invariant et correction

Notation :

- on note s pour somme
- on appelle « itération » une exécution des instructions de la boucle (du `while`)
- on note x la variable avant l'itération et x' après : s, s', i, i'...

Vérifier le Variant (par récurrence) :

- Au début i = 1 donc V = n - i + 1 = n - 1 or n > 0 (hypothèse) donc V >= 0.
- Hypothèse : V >= 0 avant l'itération, i.e. n - i + 1 >= 0
- Question : V' >= 0 et V' < V ? (V' = variant après l'itération)
  - après l'itération on a : n' = n, s' = s + i et i' = i + 1
  - V' = n' - i' + 1 (déf.) = n - (i - 1) - 1 (rempl de n' et i') = n - i
  - on est passé dans la boucle donc i <= n donc n - i >= 0 donc V' >= 0. CQFD
  - V' = n - i = (n - i + 1) - 1 = V - 1 donc V' < V donc décroissant. CQFD

Vérifier l'invariant (par récurrence) :

- Au début : i = 1 et somme = 0 donc I est vrai.
- On suppose I vrai avant une itération, après l'itération :
  - s' = s + i = $\sum_{k=0}^{i-1} k$ + i (car I vrai) = $\sum_{k=0}^{i} k$ (on intègre i)
  - or i' = i + 1 donc i = i' - 1 et donc s' = $\sum_{k=0}^{i'-1} k$. CQFD.

Vérifier la correction : Après la fin du **while** on a :

- non (i <= n) car on est sorti. Donc i > n.
- V >= 0 donc n - i + 1 >= 0 donc i <= n + 1.
  - Avec la condition précédente, on en déduit i = n + 1.
- I est vrai donc somme = $\sum_{k=0}^{i-1} k$ or i = n + 1. Donc somme = $\sum_{k=0}^{n} k$. CQFD.

Instrumenter manuellement variants et invariants

somme_n.py

```python
""" Instrumentation manuelle (avec assert) des conditions, variants et invariants"""
def somme() -> None:
    """Afficher la somme des n (10) premiers entiers."""
    n: int = 10             # un entier
    # calculer la somme des n premiers entiers   n: in; somme; out
    assert n > 0    # hypothèse sur n
    somme: int = 0        # la somme des entiers
    i: int = 1            # pour parcourir les entiers de 1 à n
    V: int = n - i + 1    # Le variant initial
    while i <= n:    # i est à prendre en compte
        # Variant : n - i + 1       # nb d'entiers restant à traiter
        assert V >= 0   # Variant positif
        # Invariant : somme =  0 + 1 + ... + (i - 1)
        assert somme == sum(range(i))  #! somme des entiers de 0 à i-1
        somme = somme + i  #! ou somme += i
        i = i + 1          #! ou i += 1
        ancien_V: int = V          # Mémoriser la valeur de V
        V = n - i + 1         # Nouveau V
        assert V < ancien_V  # Variant décroissant
    assert not (i <= n)            # sortie de la boucle
    assert V >= 0                  # Variant positif
    assert somme == sum(range(i)) # Invariant toujours vrai
    # Afficher la somme
    print(f'la somme des entiers de 1 à {n} est : {somme}')
if __name__ == "__main__": somme()
```

- Fastidieux si manuel mais des outils et techniques existent pour le faire industriellement.
- Ils sont utilisés dans les systèmes critiques (aéronautique, ferroviaire, ...).

Exercices

Exercice : Afficher les cubes consécutifs

cubes_consecutifs_while.py

Afficher les cubes des entiers de `debut` à `fin`.

Exemple : Si `debut` vaut 2 et `fin` vaut 4, le programme affiche :

```
8
27
64
```

Exercice : Cubes dont la somme est inférieure à une limite

cubes_limite_somme.py

Afficher, dans l'ordre croissant, les cubes des entiers strictement positifs à condition que leur somme soit inférieure à une limite.

Exemples :

Si `limite` vaut 30, le programme affichera :

```
1
8
```

Si `limite` vaut 36, le programme affichera :

```
1
8
27
```

Répétion **for** (PourChaque)

Forme

```
for nom in expression:
    bloc
```

Exemples

```
for nombre in range(1, 6):
    print( nombre ** 2, end=' ' )
```
affiche '1 4 9 16 25 '

```
for lettre in "XYZ":
    print(lettre, end='...' )
```
affiche 'X...Y...Z...'

Exécution

- *expression* doit être une *séquence*[a], par exemple range, list, tuple, str... (voir séquences)
- *nom* prend successivement chaque valeur de la *séquence*
- *bloc* est exécuté pour chaque affectation de *nom*

a. En fait, ce doit être un *itérable*.

Principe

- On sait combien de fois on exécute *bloc* (autant que d'éléments dans *expression*)
- La terminaison est donc garantie si la séquence est finie.

Xavier Crégut <prenom.nom@enseeiht.fr>

range : séquence d'entiers (voir help('range'))

Forme générale

- `range(debut: int, fin: int, pas:int) -> range`
- correspond aux entiers de **debut** inclus à **fin** exclu de **pas** en **pas**

Exemples

```
for nombre in range(0, 12, 3):        # range(debut, fin, pas) avec fin exclu.
    print(nombre, end=' ')
```
affiche '0 3 6 9 '

```
for nombre in range(5, -5, -2):       # Le pas peut être négatif !
    print(nombre, end=' ')
```
affiche '5 3 1 -1 -3 '

```
for nombre in range(15, 10, 2):       # Peut être vide
    print(nombre, end=' ')
```
n'affiche rien car ne passe pas dans la boucle puisque 15 est plus grand que 10

Autres formes

- `range(debut, fin)` est équivalent à `range(debut, fin, 1)` : le pas vaut 1 par défaut.
- `range(fin)` est équivalent à `range(0, fin)`

Xavier Crégut <prenom.nom@enseeiht.fr>

Exercice : Table de multiplication

table7.py

Afficher la table de multiplication de 7.

```
1 x 7 =  7
2 x 7 = 14
...
8 x 7 = 56
9 x 7 = 63
```

Exercices

cubes_consecutifs.py

Écrire avec un **for** les exercices faits avec **while**. Possible ? Plus simple ?

Exercice : **while** ou **for**

Donner des éléments pour aider à choisir entre un **while** et un **for**.

Exercice : Réécrire un for avec un while

reecrire_for_range_avec_while.py

Réécrire avec un `while` le programme suivant (on suppose pas > 0) :

```
for n in range(debut, fin, pas):
    print(n)
```

Exercice : Table de Pythagore

table_pythagore.py

Afficher la table de Pythagore, la table des multiplications en deux dimensions de 1 à un entier donné (compris entre 1 et 9) qui correspondra donc à la taille de la table.

Exemples :

Si la taille est 3 :

X	1	2	3
1	1	2	3
2	2	4	6
3	3	6	9

Si la taille est 5 :

X	1	2	3	4	5
1	1	2	3	4	5
2	2	4	6	8	10
3	3	6	9	12	15
4	4	8	12	16	20
5	5	10	15	20	25

Si la taille est 9 :

X	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9
2	2	4	6	8	10	12	14	16	18
3	3	6	9	12	15	18	21	24	27
4	4	8	12	16	20	24	28	32	36
5	5	10	15	20	25	30	35	40	45
6	6	12	18	24	30	36	42	48	54
7	7	14	21	28	35	42	49	56	63
8	8	16	24	32	40	48	56	64	72
9	9	18	27	36	45	54	63	72	81

Exercices de synthèse

Exercice : Classer un caractère

classement_caractere.py

Étant donné un caractère c (chaîne d'un seul caractère), indiquer s'il s'agit d'une voyelle, d'une consonne, d'un chiffre ou d'un autre caractère.

On ne traitera que les lettres minuscules.

Exemples :

- '0' est 'chiffre'
- 'a' est 'voyelle'
- 'c' est 'consonne'
- '!' est 'autre'
- 'A' est 'autre'

Exercice : Ordonner trois valeurs

dans_l_ordre3.py

Échanger les valeurs de trois variables entières a, b et c pour qu'on ait `a <= b <= c`.

Exemples :

```
    avant          après
a   b   c   ->  a   b   c
5   9   1   ->  1   5   9
3   1   8   ->  1   3   8
4   4   4   ->  4   4   4
```

Exercice : Nombre d'occurences (frequence) d'un chiffre

entier_frequence_un_chiffre.py

Indiquer combien un entier n a d'occurrences de 5 dans sa représentation en base 10.

Exemples :

- 123456 a 1 occurrence de 5.
- 1515 a 2 occurrences de 5.
- 555 a 3 occurrences de 5.

Exercice : Confirmation

confirmation.py

Demander à l'utilisateur de répondre soit 'oui', soit 'non' à une question. On doit lui demander de répondre à nouveau s'il répond autre chose que ces deux seules valeurs possibles.

Exemples :

L'utilisateur répond 'oui' après plusieurs erreurs :

```
Confirmation (oui/non) ? o
Merci de répondre par 'oui' ou 'non'
Confirmation (oui/non) ? NON
Merci de répondre par 'oui' ou 'non'
Confirmation (oui/non) ? oui
oui
```

L'utilisateur répond 'non' sans erreur :

```
Confirmation (oui/non) ? non
non
```

Exercice : Fréquences des chiffres dans un entier.

entier_frequence_chiffres.py

Afficher la fréquence des chiffres dans un entier.

Exemple : si l'entier est 9424549, affiche :

```
0 (0)
1 (0)
2 (1)
3 (0)
4 (3)
5 (1)
6 (0)
7 (0)
8 (0)
9 (2)
```

Exercice : Fréquences des chiffres d'un entier.

entier_frequence_chiffres_non_nulles.py

Afficher, dans l'ordre croissant des chiffres, la fréquence des chiffres d'un entier.

Exemple : si l'entier est 9424549, affiche :

```
2 (1)
4 (3)
5 (1)
9 (2)
```

Exercice : Chiffre le plus présent dans un entier

entier_frequence_chiffre_max.py

Afficher le chiffre qui a la plus grande fréquence dans un entier ainsi que sa fréquence. Si plusieurs chiffres apparaissent avec la fréquence maximale, on affichera celui le plus à gauche dans l'écriture en base 10 de l'entier.

- Si entier est 9424549 la réponse est : 4 (3 fois)
- Si entier est 100010 la réponse est : 0 (4 fois)
- Si entier est 1234567890 la réponse est : 1 (1 fois)

Sommaire

Motivation

- Cet exercice a déjà été proposé comme exercice de synthèse de la partie algorithmique.
- Il est corrigé sur les transparents suivants

Exercice (3 minutes)

On veut afficher la fréquence (nombre d'occurrences) des 10 chiffres dans un entier naturel donné.

1. Écrire un programme qui affiche la fréquence des 10 chiffres.
   Exemple : Pour l'entier 4210994, le programme affichera :
   ```
   fréquence de 0 : 1
   fréquence de 1 : 1
   fréquence de 2 : 1
   fréquence de 3 : 0
   fréquence de 4 : 2
   fréquence de 5 : 0
   fréquence de 6 : 0
   fréquence de 7 : 0
   fréquence de 8 : 0
   fréquence de 9 : 2
   ```
2. Modifier le programme pour n'afficher que les fréquences non nulles.
3. Modifier le programme pour les afficher dans l'ordre des fréquences décroissantes.

Solution naïve : Quels défauts ? Comment les corriger ?

```python
def afficher_frequences() -> None:
    '''fréquence des chiffres d'un nombre'''

    # saisir un nombre (sans contrôle)
    nombre: int = int(input('Un entier naturel : '))

    # calculer les fréquences des chiffres
    frequence0: int = 0   # fréquence du chiffre 0
    frequence1: int = 0   #     ''          ''     1
    frequence2: int = 0   #     ''          ''     2
    frequence3: int = 0   #     ''          ''     3
    frequence4: int = 0   #     ''          ''     4
    frequence5: int = 0   #     ''          ''     5
    frequence6: int = 0   #     ''          ''     6
    frequence7: int = 0   #     ''          ''     7
    frequence8: int = 0   #     ''          ''     8
    frequence9: int = 0   #     ''          ''     9
    fini: bool = False
    while not fini:
        # comptabiliser l'unité du nombre
        unite: int = nombre % 10
        if unite == 0:
            frequence0 += 1
        elif unite == 1:
            frequence1 += 1
        elif unite == 2:
            frequence2 += 1
        elif unite == 3:
            frequence3 += 1
        elif unite == 4:
            frequence4 += 1
        elif unite == 5:
            frequence5 += 1
        elif unite == 6:
            frequence6 += 1
        elif unite == 7:
            frequence7 += 1
        elif unite == 8:
            frequence8 += 1
        else: # unite == 9
            frequence9 += 1

        # supprimer l'unité du nombre
        nombre = nombre // 10

        fini = nombre == 0

    # afficher les fréquences
    print('fréquence de 0 :', frequence0)
    print('fréquence de 1 :', frequence1)
    print('fréquence de 2 :', frequence2)
    print('fréquence de 3 :', frequence3)
    print('fréquence de 4 :', frequence4)
    print('fréquence de 5 :', frequence5)
    print('fréquence de 6 :', frequence6)
    print('fréquence de 7 :', frequence7)
    print('fréquence de 8 :', frequence8)
    print('fréquence de 9 :', frequence9)
```

Critique de la solution

Positif : Le programme fonctionne !

Négatif :

- De nombreuses redondances !
  - initialisation des variables frequence0 à frequence9
  - tous les `elif` de la boucle `while`
  - l'affichage des fréquences : 10 fois presque la même chose
- Difficulté à prendre en compte les évolutions (questions 2 et 3)
  - pour n'afficher que les fréquences non nulles, il faut rajouter 10 fois presque le même `if`
  - afficher les fréquences dans l'ordre croissant serait très fastidieux !

Comment éviter la redondance constatée ?

- Il faudrait pouvoir jouer sur le chiffre, 0 . . . 9, des noms `frequence0` . . . `frequence9`
  - C'est ce que permet le type liste (`list`) de Python
  - Une liste est une juxtapositions d'objets repérés par un indice (entier), 0 pour le premier
  - `frequences = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0] # une liste de 10 entiers valant 0`
  - `frequences` regroupe nos 10 variables : `frequences[0]` correspond à `frequence0`. . .
  - l'indice est n'importe quelle expression entière : on peut donc le calculer
  - plus court et lisible pour créer cette liste : `frequences = [0] * 10`
- On peut alors simplifier le programme en calculant l'indice

```
# comptabiliser l'unité
frequences[unite] += 1
```

```
# afficher les fréquences
for chiffre in range(0, 10):
    print('fréquence de', chiffre, ':', frequences[chiffre])
```

Les tableaux (listes) à la rescousse !

```python
def afficher_frequences() -> None:
    '''fréquence des chiffres d'un nombre'''

    # saisir un nombre (sans contrôle)
    nombre: int = int(input('Un entier naturel : '))

    # calculer les fréquences des chiffres
    frequences: list[int] = [0] * 10       # frequences[i] = fréquence du chiffre i
    fini: bool = False      # tous les chiffres de nombre traités ?
    while not fini:
        # comptabiliser l'unité du nombre
        unite: int = nombre % 10    # unité de nombre
        frequences[unite] += 1

        # supprimer l'unité du nombre
        nombre = nombre // 10

        fini = nombre == 0

    # afficher les fréquences
    for chiffre in range(10):
        print('fréquence de', chiffre,
              ':', frequences[chiffre])
```

- Pas de redondance
- Le programme est passé de 58 à 24 lignes !
- L'algorithme est conservé
- Question 2 : Ajouter un `if` dans le dernier `for`
- Question 3 : On pourrait trier sur les fréquences

Mais on pouvait le faire sans liste !

```python
def afficher_frequences() -> None:
    '''fréquence des chiffres d'un nombre'''

    # saisir un nombre (sans contrôle)
    nombre: int = int(input('Un entier naturel : '))

    # calculer les fréquences des chiffres
    for chiffre in range(10):
        # Calculer la fréquence de chiffre
        copie: int = nombre     # on va exploiter plusieurs fois nombre !
        frequence: int = 0      # fréquence de chiffre dans copie
        fini: bool = False
        while not fini:
            # comptabiliser l'unité de copie
            unite: int = copie % 10
            if unite == chiffre:
                frequence += 1

            # supprimer l'unité de copie
            copie = copie // 10

            fini = copie == 0

        # afficher les fréquences
        print('fréquence de', chiffre,
              ':', frequence)
```

- **Positif :**
  - Code redondant aussi éliminé (25 lignes).
  - Moins de mémoire utilisée (/ version liste)
    1 compteur au lieu de 10
- **Négatif :**
  - Plusieurs parcours des chiffres du nombre (cf copie)
    Donc plus coûteux en temps que la version liste
  - Ne permet pas de répondre à la question 3
- **Conclusion :**
  - Compromis espace mémoire / temps calcul !
  - **Ne jamais écrire de code redondant !**

Motivation (suite)

Exercice (1 minute) : Afficher dans l'ordre croissant une série de valeurs saisies

Afficher dans l'ordre croissant une série de valeurs réelles saisies au clavier. La série se termine par la valeur zéro qui ne fait pas partie de la série.
Voici quelques exemples :

```
1 2 3 0        --> 1 2 3
3 2 1 0        --> 1 2 3
5 2 3 8 -1 0   --> -1 2 3 5 8
```

Premier niveau de raffinage

```
R0 : Afficher une série saisie dans l'ordre croissant

R1 : Raffinage De « Afficher une série saisie dans l'ordre croissant »
    | Saisir la série                  valeurs: out
    | Trier les valeurs de la série    valeurs: in out
    | Afficher les valeurs              valeurs: in
```

Question : Quel type prendre pour valeurs ?
Réponse : Forcément plusieurs éléments. Par exemple, une liste de réels !
Contrairement à l'exemple précédent, il faut conserver en mémoire les valeurs de la série !

Les trois actions de R1 constituent des problèmes récurrents.

Séquence

Définition

Une **séquence** est un type de données qui permet de regrouper dans un même objet un nombre fini d'objets.
Ces objets sont repérés par leur **position** (aussi appelée **indice** ou **index**).

En Python

Plusieurs types de séquences existent :

1. des séquences modifiables : liste (list)
2. des séquences non modifiables : n-uplet (tuple), chaîne (str), intervalle (range)

Exemples

```
s = [4, 'x', 2, False, 2, 7]  # une liste (list)
c = 'bonjour'                 # une chaîne (str)
t = ('a', 1, 2.5)             # un n-uplet (tuple)
r = range(1, 10, 2)           # un intervalle (range)
```

- Les **éléments** d'une séquence peuvent être de types différents (typage dynamique).
- Le même élément peut apparaître plusieurs fois (cas de 2 dans s).
- Une séquence peut être vide : `[]` `''` `""` `()` `range(5, 1)`

Xavier Crégut <prenom.nom@enseeiht.fr>

Taille et indices

Taille d'une séquence : `len(s)`

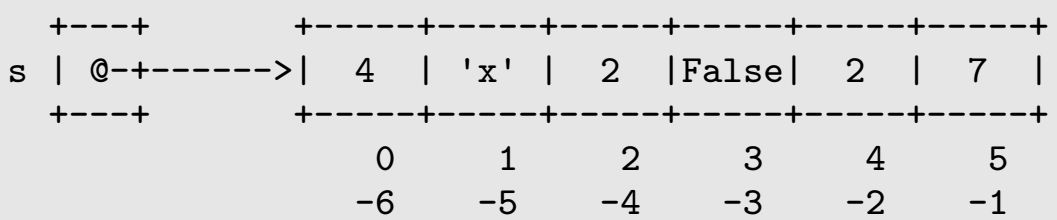
La fonction len() donne la taille de la séquence s fournie en paramètre.
C'est le nombre d'éléments que la séquence contient.

Exemples

```
assert len(s) == 6
assert len(c) == 7
assert len(t) == 3
assert len(r) == 5
assert len([]) == 0
```

Indices

- **Définition :** Un indice permet de désigner un emplacement d'une séquence : `s[i]`
- Un **indice valide** sur une séquence s est un entier i tel que `-len(s) <= i < len(s)`
  - Les indices positifs repèrent les élements de la gauche vers la droite (0 le plus à gauche)
  - Les indices négatifs repèrent les éléments de la droite vers la gauche (-1 le plus à droite)




```
len(s) == 6
s[0] == s[-6] == 4
s[5] == s[-1] == 7
```

- Premier élément (le plus à gauche) : s[0] (ou s[-len(s)], moins pratique)
- Dernier élément (le plus à droite) : s[-1] (ou s[len(s) - 1], moins pratique)
- Si l'indice n'est pas valide, l'exception `IndexError` est levée.

Opérations élémentaires sur les séquences modifiables

- Préambule : Tous les exemples sont exécutés avec `s = [4, 'x', 2, False, 2, 7]`.

Principe : s[i] est équivalent à une variable (nom) qui désigne l'emplacement d'indice i de s.

- `s[indice]` : accès à l'élément à la position `indice` de la séquence s
  - `assert s[0] == 4`
  - `v = s[6]` : lève l'exception `IndexError`
- `s[indice] = expression` : remplacer un élément d'une séquence...
  - associer à `s[indice]` la valeur de `expression`
  - l'élément à la position indice de la séquence est donc la valeur de l'expression
  - `s[3] = 2 ** 3;  assert s[3] == 8 ; assert s == [4, 'x', 2, 8, 2, 7]`
  - `s[0] = s[0] + 1; assert s[0] == 5 ; assert s == [5, 'x', 2, False, 2, 7]`
  - on parle de *left value* (à gauche de l'affectation, ~ variable) et *right value* (à droite, ~ expression)

- `s.append(expression)` : ajouter la valeur de expression à la fin de la séquence s
  - append est une méthode, d'où la notation pointée.
  - `s.append(5);  assert s == [4, 'x', 2, False, 2, 7, 5];  assert s[-1] == 5`
  - Attention : append retourne None (instruction) : `r = s.append(5);  assert r == None`
- `s.pop(indice)` : supprimer l'élément à un indice donné
  - `s.pop(1) ; assert s == [4, 2, False, 2, 7];  assert s[1] == 2`
  - retourne l'élément supprimé : `r = s.pop(1) ; assert r == 'x'`
  - alternative : `del s[indice]` (pop est à préférer)

Xavier Crégut <prenom.nom@enseeiht.fr>

Exercices

Exercice : Trouver le type

Indiquer le type des objets suivants :

- `(1.5, 2, "dix")`
- `'liste'`
- `range(3, 10, 2)`
- `[1.5, 2, "dix"]`

test_sequence_exemple_cours.py

```python
# Remplacer les ... pour que le programme s'exécute
# sans aucune erreur signalée par les assert

s = [10, 3, 4, 7, 3, 5]

taille = ...          # la taille de s

assert taille == 6

premier = ...       # le premier élément de s
dernier = ...       # le dernier élément de s

assert premier == 10
assert dernier == 5

indice7 = ...       # entier qui correspond à l'indice de 7 dans s

assert s[indice7] == 7

...                 # remplacer 4 par 421 dans s

assert s == [10, 3, 421, 7, 3, 5]

...                 # supprimer 421 de s

assert s == [10, 3, 7, 3, 5]

...               # ajouter 0 à la fin de s

assert s == [10, 3, 7, 3, 5, 0]
print('ok')
```

Exploiter les éléments d'une séquence

Affectation multiple pour destructurer une séquence

On peut initialiser plusieurs noms avec une séquence.

```
a, b, c = [1, 2, 3]          ; assert a == 1 and b == 2 and c == 3
```

Il doit y avoir autant de noms que d'objets dans la séquence

```
a, b = [1, 2, 3]     # ValueError: too many values to unpack (expected 2)
a, b, c = [1, 2]     # ValueError: not enough values to unpack (expected 3, got 2)
```

On peut avoir un nom préfixé de *. Il correspond à une liste et absorbe les éléments en surplus

```
p, *m, d = [1, 2, 3, 4]    ; assert p == 1 and m == [2, 3] and d == 4
p, *m, d = [1, 2]          ; assert p == 1 and m == [] and d == 2
```

Mais il faut assez d'éléments à droite et au plus une * à gauche.

```
p, *m, d = [1] # ValueError: not enough values to unpack (expected at least 2, got 1)
p, *m, d, *n = [1, 2, 3, 4] # SyntaxError: two starred expressions in assignment
```

for et séquence

On peut utiliser un **for** avec une séquence :

```
for x in s:     #! x est associé successivement à chaque objet de la séquence s
    print(x)
```

Exercice : Somme des cubes

sequence_somme_cubes.py

Calculer la somme des cubes des éléments d'une séquence d'entiers.

Exemples :

```
séquence            somme
[ 4, -2, 0]    ->   56
[]             ->   0
(1, 2, 3, 4)   ->   100
```

Exercice : Destructuration d'une séquence

test_sequence_destructuration.py

```
# Remplacer les ... par une seule instruction
# sans écrire de constantes littérales (1, 2...)
# Le programme doit s'exécute sans aucune erreur
s = [1, 2, 3, 4]

...

assert a0 == 3 and b0 == 2 and c0 == 1 and d0 == 4

...

assert a1 == 1 and b1 == [2, 3, 4]

...

assert a2 == 4 and b2 == 1 and c2 == [2, 3]

...

assert a3 == 4 and b3 == [1, 2] and c3 == 3

print('ok')
```

Exercice (6 minutes)

D'après les deux exemples suivants (4 programmes), quelle forme du `for` préférer :

1. `for` sur les éléments (programmes de gauche) ou
2. `for` sur les indices (programmes de droite)

Exemple : Calculer la somme des élements d'une liste d'entiers

Exemple : si liste == [4, 6, 1, 9, 3] alors somme == 33

```
sequence = ... # à définir
somme = 0
for element in sequence:
    somme += element
```

```
sequence = ... # à définir
somme = 0
for indice in range(len(sequence)):
    somme += sequence[indice]
```

Exemple : Calculer la somme des élements d'indice pair d'une liste d'entiers

Exemple : si liste == [4, 6, 1, 9, 3] alors somme == 8

```
sequence = ... # à définir
somme = 0
indice = 0
for element in sequence:
    if indice % 2 == 0:
        somme += element
    indice += 1
```

```
sequence = ... # à définir
somme = 0
for indice in range(len(sequence)):
    if indice % 2 == 0:
        somme += sequence[indice]
```

enumerate

Quelle forme du `for` préférer ?

La réponse est le `for` sur les éléments car :

- la ligne du `for` est plus simple (pas de `range`)
- l'accès à l'élément est direct donc plus lisible (pas d'indice) : `element` vs `sequence[i]`

Elle comporte un défaut quand on doit manipuler l'indice : il faut le gérer explicitement !

enumerate

enumerate(s, start) où s est une séquence et start un entier (0 si non précisé)
crée une séquence de couples de la forme (start, s[0]), (start+1, s[1])...
Par exemple, `enumerate([4, -2, 0])` crée l'équivalent de `[(0, 4), (1, -2), (2, 0)]`.

Grâce à enumerate, la version `for` sur élément redevient plus lisible :

```
sequence = ... # à définir
somme = 0
for indice, element in enumerate(sequence):
    if indice % 2 == 0:
        somme += element
```

```
sequence = ... # à définir
somme = 0
for indice in range(len(sequence)):
    if indice % 2 == 0:
        somme += sequence[indice]
```

1. Comment comprendre `for indice, element in enumerate(sequence):` ?
2. Y a-t-il des cas où il faut utiliser `for` avec indices ?

Xavier Crégut <prenom.nom@enseeiht.fr>

Comment comprendre `for indice, element in enumerate(sequence):` ?

- Il s'agit d'une destructuration d'une séquence
- En effet, enumerate est une séquence de couples
- On fait un `for` dessus
- On fait donc `indice, element = couple`
- Donc indice correspond au premier élément (le numéro de séquence)
- et element au deuxième (un élément de la séquence)

On aurait pu l'écrire ainsi (qui rend la destructuration explicite) :

```
for couple in enumerate(sequence):
    indice, element = couple
```

Le `for` avec indice est utile !

Exercice (3 minutes)

1. Est-ce que les deux programmes suivants répondent à l'énoncé ?
2. Qu'en conclure sur l'utilisation de `for` ?

Exemple : RAZ

Remplacer tous les éléments d'une liste par 0

Exemples :

```
séquence avant        séquence après
[ 4, -2, 0]      ->   [0, 0, 0]
[]               ->   []
['a', True]      ->   [0, 0]
```

```python
sequence = ... # à définir
for element in sequence:
    element = 0
```

```python
sequence = ... # à définir
for indice in range(len(sequence)):
    sequence[indice] = 0
```

Réponse

Constatation :

- Le programme de gauche laisse la liste inchangée.
- Le programme de droite donne le bon résultat.

Pourquoi ?

- `sequence[indice] = 0` change bien l'élément de la liste par 0
- `element = 0` change l'objet associé à `element` mais ne change pas la liste
  - C'est comme quand on fait

```
x = 5     # équivalent de sequence[indice] (un élément de la liste)
y = x     # équivalent de element (un nom qui référence un élément de la liste)
y = 0     # Cette affectation change la valeur de y, pas celle de x
```

Conséquence :

- Toujours utiliser l'indice si on veut changer un élément de la liste
  - Remarque : l'indice peut être obtenu par enumerate.

Conclusion

1. Préférer le `for` sur les éléments
   - il est plus général et marchera avec d'autres types que les séquences
2. Sauf si on doit s'arrêter avant d'avoir parcouru tous les éléments de la séquence
3. Utiliser un indice (via range ou enumerate) pour changer un élément de la séquence

break et continue

Les `for` et `while` de Python acceptent les instructions `continue` et `break` qu'il est interdit d'utiliser pour l'instant.

Exercice : Nombre d'occurrences d'un élément dans une séquence. sequence_frequence.py

Calculer le nombre d'occurrences de x dans une séquence s.

Exemples :

- si s = [1, 3, 5, 3] et x = 3 alors le résultat est 2
- si s = [1, 3, 5, 3] et x = 1 alors le résultat est 1
- si s = (3, 2, 5, 5, -5) et x = 0 alors le résultat est 0
- si s = 'dernier' et x = 'e' alors le résultat est 2

Exercice : Est-ce qu'un élément est présent dans une séquence ? sequence_est_present.py

1. Déterminer si un élément x est présent dans une séquence s.
   Exemples :
   - si s = [1, 3, 5, 0] et x = 5 alors la réponse est oui
   - si s = [1, 3, 5, 0] et x = 4 alors la réponse est non
   - si s = [1, 3, 5, 0] et x = 0 alors la réponse est oui
   - si s = [1, 3, 5, 0] et x = 1 alors la réponse est oui
2. Dans le dernier exemple, combien le programme fait de comparaisons sur les éléments de la séquence avant de donner le résultat.
3. Il faudrait que le programme arrête de parcourir la séquence dès qu'il a trouvé l'élément. Modifier le programme si ce n'est pas le cas.

Exercice : Remplacer toutes les occurrences d'un élément par un autre

sequence_remplacer.py

1. Remplacer dans une séquence s toutes les occurrences de x par n (comme nouveau).
   Exemples :
   - si s = [5, 3, 8, 2, 3, 1], x = 3 et n = 0 alors s devient [5, 0, 8, 2, 0, 1]
   - si s = [], x = 3 et n = 0 alors s devient []
2. Ce programme fonctionne-t-il avec un n-uplet (tuple) ou une chaîne (str) ?

Exercice : Indice d'un élément dans une séquence

sequence_indice_element.py

Déterminer l'indice d'un élément x dans une séquence s. Si l'élément est présent plusieurs fois, on donnera l'indice le plus petit. Si l'élément n'est pas trouvé, on répondra None.

Exemples :

- si s = [5, 3, 8, 2, 3, 1] et x = 8 alors l'indice est 2
- si s = [5, 3, 8, 2, 3, 1] et x = 3 alors l'indice est 1
- si s = [5, 3, 8, 2, 3, 1] et x = 9 alors l'indice est None

Opérations classiques sur une séquence

Les opérations élémentaires du transparent précédent sont suffisantes mais d'autres opérations de plus haut niveau seraient bien utiles :

- savoir si un élément est présent dans une séquence (`in`)
- supprimer un élément d'une séquence en précisant sa position (`pop`)
- supprimer un élément d'une séquence (`remove`)
- compter le nombre d'occurrences d'un élément d'une séquence (`count`)
- insérer un nouvel élément à un indice donné (`insert`)
- remplacer toutes les occurrences d'un élément par un autre élément (`replace`)
- obtenir l'indice de la première occurrence d'un élément dans une séquence (`index`)
- trier les éléments d'une séquence dans l'ordre croissant (`sort`)
- concaténer à une séquence une autre séquence (ajouter à la fin) (`extend`)
- savoir si deux séquences sont égales (même type, même taille et mêmes éléments) (`==`)
- ...

Remarque : Ces opérations sont déjà disponibles en Python (nom entre parenthèses).

- On peut créer une liste ou un n-uplet à partir d'une séquence :

```
t = tuple('texte')        ; assert t == ('t', 'e', 'x', 't', 'e')
l = list(t)               ; assert l == ['t', 'e', 'x', 't', 'e']
r = list(range(2, 9, 2)) ; assert r == [2, 4, 6, 8]

s = ''.join(l)            ; assert s == 'texte'  # '' séparateur pour lier les chaînes
c = str(100)              ; assert c == '100'    # str(o) : la chaîne présentant l'objet o
```

Concepts avancés

Pourquoi « concepts avancés » ?

1. Parce qu'on peut écrire des programmes sans les connaître
   - C'est ce qu'on a fait jusqu'à présent !
2. Mais qu'il est important de les comprendre pour savoir ce qu'on fait
   - Il est important de bien comprendre le fonctionnement d'un langage sinon
   - on peut se retrouver avec un programme dont on ne comprend pas le fonctionnement
   - Ces notions seront importantes avec les sous-programmes
   - Mais on peut rencontrer ces problèmes dès maintenant

Exercice (2 minutes)

1. Après lecture des deux programmes suivants, dire ce qui devrait s'afficher.
2. Exécuter ces programmes et vérifier les réponses précédentes.

```
x1 = 421
x2 = x1
x2 += 1
print(x1)
```

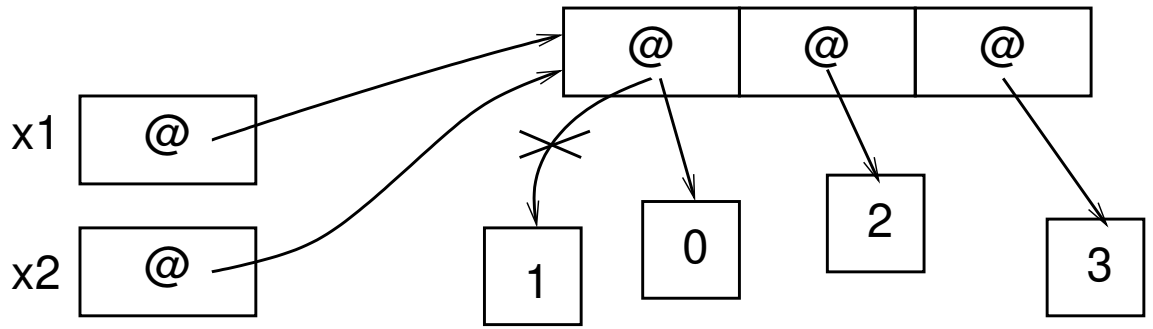
```
x1 = [1, 2, 3]
x2 = x1
x2[0] = 0
print(x1)
```

Bien comprendre Nom et Objet : le partage

- Un objet ne peut changer ni d'identité, ni de type.
- Plusieurs noms peuvent référencer le même objet (ce sont des **alias**).
- Tout changement fait sur l'objet depuis l'un des noms est visible depuis les autres noms !

```
1 x1 = [1, 2, 3]      # une liste contenant 3 éléments 1, 2 et 3
2 x2 = x1             # un deuxième nom sur la même liste
3 x1[0] = 0           # changement du premier élément de la liste
4 print(x2)           # [0, 2, 3]
5 id(x1) == id(x2)    # True : x1 et x2 donnent accès au même objet
```

- Un nom est bien un **accès**, **pointeur**, **référence** sur un objet



- **Important :** L'affectation associe un objet à un nom. Elle ne copie pas l'objet !
- Exécuter ce progamme avec Python tutor :
  - même représentation sauf pour les types simples (entier, réel...) où la flèche n'est pas représentée

Xavier Crégut <prenom.nom@enseeiht.fr>

Égalité physique (is) et égalité logique (==)

- **Égalité physique :** deux noms référencent le même objet (mêmes identifiants)
- **Égalité logique :** deux objets, éventuellement (d'identifiants) différents, ont mêmes valeurs

```
1  x1 = [1, 2, 3]  # une liste contenant 3 éléments 1, 2 et 3
2  x2 = x1         # un deuxième nom sur la même liste
3  x3 = [1, 2, 3]  # une autre liste contenant 1, 2 et 3
4  x1 is x2     # True        id(x1) == id(x2)
5  x1 is x3     # False       id(x1) != id(x3)
6  x1 == x2     # True        même nombre d'éléments et mêmes éléments
7  x1 == x3     # True                  "           "            "
8  x3[0] = 0    #             on modifie l'objet associé à x3
9  x1 == x3     # False       les deux listes sont différentes (premiers éléments différents)
10 x1 != x3     # True, négation de ==
11 x1 is not x3 # True, négation de is
```

- L'opérateur **is** teste l'**égalité physique** : même objet
- L'opérateur == teste l'**égalité logique** : mêmes valeurs
- `n1 is not n2` est équivalent à `not (n1 is n2)`
- `n1 != n2` est équivalent à `not (n1 == n2)`
- **Normalement :** égalité physique implique égalité logique (exception : NaN, i.e. math.nan)

Remarque : `n1 is n2` est équivalent à `id(n1) == id(n2)`. Préférer `is` !

Xavier Crégut <prenom.nom@enseeiht.fr>

Objet muable et objet immuable

Définition :

- Objet **immuable** : objet dont l'état ne peut pas changer après sa création.
- Objet **muable** : objet dont l'état peut changer au cours de sa vie.
- Synonymes : altérable/inaltérable, modifiable/non modifiable (anglais : *mutable/imutable*)

```
1  s1 = 'bon'           # Les chaînes sont immuables :
2  s1[0] = 'B'          # TypeError: 'str' object does not support item assignment
3  del s1[0]            # TypeError: 'str' object doesn't support item deletion
4  s2 = s1.lower()      # Création d'un nouvel objet
5  s2 is s1             # False            on a bien deux objets différents
6  s2 == s1             # True             mais logiquement égaux
7
8  l1 = [ 1, 2, 3 ]     # Les listes sont muables, la preuve :
9  l1[0] = -1                    # [-1, 2, 3]
10 del l1[0]                     # [2, 3]
11 l1.append(4)                  # [2, 3, 4]
```

Objet immuable vs objet muable :

- Un objet immuable peut être partagé sans risque car personne ne peut le modifier
- Mais chaque « modification » nécessite la création d'un nouvel objet ! (exemple : str.lower())

Chaînes litérales

- Utiliser apostrophe (') ou guillemet (") pour délimiter les chaînes

```
1   s1 = "Bonjour"       # avec des guillemets
2   s2 = 'Bonjour'       # avec apostrophe
3   s3 = 'Bon' "jour"    # concaténation implicite
4   s4 = 'Bon' + "jour" # concaténation explicite
5   s5 = 'Bon'  \        # \ : caractère de continuation
6        "jour"
7   s6 = ('Bon'              # caractère \ inutile si (, [ ou { ouvert
8        "jour")
9   s7 = """Une chaîne
10       sur plusieurs
11       lignes"""
12  s8 = '''Avec une ' (apostrophe)
13       dedans'''
14  s9 = "des caractères spéciaux : \t, \n, \", \'..."
15  sA = 'valeur : ' + str(12)  # conversion explicite en str()
```

- Variante avec **triple délimiteur** : permet de continuer la chaîne sur plusieurs lignes.
- Les chaînes avec triple délimiteurs sont utilisées pour la **documentation**.
- Il n'y a **pas de type caractère** en Python (idem chaîne de longueur 1)

Opérateurs sur str (comme séquence immuable)

Opération	Résultat	Exemples
x in s	True si x est une sous-chaîne de s	'bon' in 'bonjour'
x not in s	True si x n'est pas une sous-chaîne de s	'x' not in 'bon'
s + t	concaténation de s avec t	
s * n ou n * s	équivalent à ajouter s à elle-même n fois	'x' * 3 donne 'xxx'
s[i]	ième caractère de s, i == 0 pour le premier	'bon'[-1] donne 'n'
len(s)	la longueur de s (nombre de caractères)	len('bon') donne 3
min(s)	plus petit caractère de s	min('onjour') donne 'j'
max(s)	plus grand caractère de s	max('onjour') donne 'u'
s.index(x[, d[, f]])	indice de la première occurrence de x dans s (à partir de l'indice d et avant l'indice f)	'bonjour'.index('o') donne 1 'bonjour'.index('o', 2) donne 4
s.count(x)	nombre total d'occurrence de x dans s	'bonjour'.count('o') donne 2

- i est un indice valide sur s ssi -len(s) $\leq$ i $<$ len(s), sinon IndexError !
- `'bonjour'.index('o', 2, 4)` lève l'exception ValueError car non trouvé
- **Remarque :** Toutes ces opérations sont présentes sur toute séquence.
- Les chaînes de caractères sont des **séquences immuables** de caractères.

Méthodes spécifiques de str

```
s = ' ET '.join( ['un' , 'deux', 'trois'] ) # concaténer avec ce séparateur
assert s == 'un ET deux ET trois'

assert 'chat' < 'chien'    # (Ordre lexicographique, celui du dictionnaire)
assert 'chat' < 'chats'

code = ord('0')       # ord: obtenir le code d'un caractère
assert code == 48     # ... mais quel intérêt de connaître sa valeur précise ? Aucun !
c = chr(code)         # chr : obtenir le caractère correspondant à un code
assert c == '0'
                            # Intérêt de ord et chr :
c = '5'                     # - passer d'un chiffre caractère à l'entier corrspondant
chiffre = ord(c) - ord('0')
assert chiffre == 5
chiffre = 9                 # - et inversement
c = chr(ord('0') + chiffre)
assert c == '9'

r = 'bonjour'.replace('o', 'OO')   # remplacer toutes les occurrences de 'o' par 'OO'
assert r == 'bOOnjOOur'

r = '   xx yy   zz  '.strip()  # supprimer les blancs du début et de la fin
assert r == 'xx yy  zz'

r = '   xx yy   zz  '.split()  # découper une chaîne en liste de chaînes
assert r == ['xx', 'yy', 'zz']
```

Mais aussi lower, islower, upper, isupper, isdigit, isalpha, etc. Voir help('str').

Exercices

1. Comment obtenir le dernier caractère d'une chaîne ?
   - Exemple : 'bonjour' -> 'r'
2. Remplacer tous les `'e'` d'une chaîne par `'*'`.
   - Exemple : `'une chaîne'` -> `'un* chaîn*'`
3. Idem avec `'ne'` deviennent `'...'`.
   - Exemple : `'une chaîne'` -> `'u... chaî...'`
4. Étant donné une chaîne et un caractère, trouver la position de la deuxième
   - occurrence de ce caractère dans la chaîne.
   - Exemple : 'bonjour vous' et 'o' -> 4
5. Indiquer combien il y a de mots dans une chaîne de caractères.
   - Exemple : 'bonjour vous' -> 2
   - Exemple : ' il fait très beau ' -> 4
6. Indiquer le nombre d'occurrences d'une lettre dans une chaîne.
   - Exemple : "C'est l'été, n'est-ce pas ?" contient 1 'a', 3 'e', 0 'v', 3 '', 2 'st', etc.

N-uplet (tuple) : séquence immuable

- **Définition :** Un n-uplet (tuple) est une séquence immuable d'objets quelconques

```
t = (1, 'deux', 10.5)   # t est un tuple composé de 3 objets
u = 1, 'deux', 10.5     # les parenthèses peuvent être omises (si pas ambigu)
v = (1, )   # tuple avec un seul élément (virgule obligatoire)

a, b, c = t # destructurer le tuple : a == 1, b == 'deux', c == 10.5
_, x, _ = t # x == 'deux' et _ == 10.5 (_ pour dire que l'on ne s'en servira pas)
t[0]        # 1
t[-1]       # 10.5
```

- Représenter une date avec le numéro du jour, du mois et de l'année :

```
date_v3 = (3, 12, 2008) # ici, on choisit (jour, mois, année). À documenter !
j, m, a = date_v3       # retrouver ses constituants
```

- On peut construire un tuple à partir d'une séquence

```
tuple('abc')    # ('a', 'b', 'c')
tuple([1, 'X']) # (1, 'X')
```

- Le tuple n'est pas modifiable... mais ses objets peuvent l'être

```
t = (1, [1])   # t référencera toujours cet entier et cette liste
t[1][0] = 2    # L'objet liste est modifiable !
assert t == (1, [2])   # C'est le même objet liste... qui a changé
t[1] = []      # TypeError: 'tuple' object does not support item assignment
```

Liste (list) : séquence modifiable

Création d'une liste

```
l1 = [ 1, 2, 3]       # liste [1, 2, 3]
l2 = []               # une liste vide
```

Opérations sur une liste

```
l1[0]           # 1 : premier élément de la liste
len(l1)         # 3
p, *m, d = l1   # p == 1 and m == [2] and d == 3

l1 += [4, 5]    # l1 == [1, 2, 3, 4, 5]     # concaténer (ou l1.extend([4, 5])), même liste
l1 = l1 + [6]   # l1 == [1, 2, 3, 4, 5, 6]  # mais création d'une nouvelle liste !
del l1[3]       # l1 == [1, 2, 3, 5, 6]
l1[1] = 'x'     # l1 == [1, 'x', 3, 5, 6]         # hétérogène !
l1.append('x')  # l1 == [1, 'x', 3, 5, 6, 'x']    # ajouter un élément à la fin
l1.count('x')   # 2                               # nombre d'occurrences d'un élément
l1.insert(1, 9) # l1 == [1, 9, 'x', 3, 5, 6, 'x'] # insérer l'élément à l'indice
l1.remove('x')  # l1 == [1, 9, 3, 5, 6, 'x'] # supprimer la première occurrence
x = l1.pop(1)   # x == 9 and l1 == [1, 3, 5, 6, 'x']  # supprime l'élément à indice
```

Séquence modifiable

Une liste est une séquence modifiable. Elle a donc toutes les opérations d'une séquence immuable + des opérations de modification.

Xavier Crégut <prenom.nom@enseeiht.fr>

Questions

Étant donnée une séquence s, par exemple s = [10, 5, 7, 15, 7, 1]

1. Comment obtenir le dernier élément ?
   - Le premier élément de s est 10.
2. Comment obtenir le dernier élément ?
   - Le dernier élément de s est 1.
3. Quel est l'indice de x dans s ?
   - L'indice de 15 dans s est 3.
4. Comment obtenir le nombre d'occurrences (la fréquence) d'un élément x ?
   - La fréquence de 7 dans s est 2.
5. Comment obtenir l'indice de la première occurrence de x dans s ?
   - L'indice de la première occurrence de 7 dans s est 2.
6. Comment obtenir l'indice de la deuxième occurrence de x dans s ?
   - L'indice de la deuxième occurrence de 7 dans s est 4.
7. Comment savoir si x est dans s ?
   - 7 est un élément de s. 11 n'est pas un élément de s. . . .

Exercices

Indiquer, après l'exécution de chaque ligne, la valeur de la liste s.

```
1  s = []
2  s.append(2)
3  s.insert(0, 4)
4  s.insert(2, 1)
5  s[1] = 'deux'
6  s[2] /= s[2]
7  s.count(1)
8  s[0], s[1] = s[1], s[0]
9
10 p, _, d = s              # p ? _ ? d ?
11 premier, *suite = s      # premier ? suite ?
12
13 b = [False, True]
14 s += b
15 s2 = [2, 3, 5]
16 i, s2[i], x = s2         # s2 ? i ? x ?
17 s.append(s2)
18 s2.append(s)             # s2 ?  print(s2) ?
19
20 s = list('Fin.')         # s ?  s2 ?
```

Xavier Crégut <prenom.nom@enseeiht.fr>

Intervalle (range) : séquence immuable d'entiers

- range permet de définir des séquences immuables d'entiers.
- Il est généralement utilisé pour les répétitions (`for`).
- Appels possibles :

```
range(start, stop[, step]) -> range object      # step == 1 si non fourni
range(stop) -> range object                     # start == 0 and step == 1
```

- construit la séquence d'entiers de start inclus à stop exclu de step en step
- Quelques exemples :

```
r1 = range(4)
r1                  # range(0, 4)
list(r1)            # [0, 1, 2, 3]
tuple(r1)           # (0, 1, 2, 3)
r1[-1]              # 3

list(range(4, 8))           # [4, 5, 6, 7]
list(range(8, 4))           # []
list(range(2, 10, 3))       # [2, 5, 8]
list(range(5, 2, -1))       # [5, 4, 3]
list(range(2, 3, -1))       # []
```

Les slices (tranches)

Motivation : Référencer une partie des objets contenus dans une séquence.

Forme générale : `sequence[debut:fin:pas]`

- les élements de `sequence` de l'indice `debut` inclu, à l'indice `fin` exclu avec un `pas`
- les indices peuvent être positifs (0 pour le premier élément) ou négatifs (-1 pour le dernier)
- possibilité d'utiliser les opérateurs classiques : **del**, =, etc.

Exemples :

```
s = list(range(10))      # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
s[2:4]                   # [2, 3]
s[2:]                    # [2, 3, 4, 5, 6, 7, 8, 9]
s[:-4]                   # [0, 1, 2, 3, 4, 5]
s[2::4]                  # [2, 6]
s[::-1]                  # [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
s[8:2:-2]                # [8, 6, 4]                (parcours de la fin vers le début)

del s[2::2]              # [0, 1, 3, 5, 7, 9]    (supprimer les éléments de la slice)
s[-4:] = s[:4]           # [0, 1, 0, 1, 3, 5]
s[:-1] = s[1:]           # [1, 0, 1, 3, 5, 5]
s[1:] = s[:-1]           # [1, 1, 0, 1, 3, 5]
s[1:3] = [9, 8, 7, 6]    # [1, 9, 8, 7, 6, 1, 3, 5]
s[::2] = list(range(9)) # ValueError: attempt to assign sequence of size 9 to
                         # extended slice of size 4
s[1:-2] = []             # [1, 3, 5]
range(0, 4)[::-1]        # range(3, -1, -1)
```

Pour en savoir plus...

Exercices

1. Étant donnée une liste L, comment obtenir la liste de tous les éléments sauf le premier et le dernier ?
   - Exemple : L = [2, 3, 4, 5] donne [3, 4]
2. Étant donnée une liste L, comment obtenir deux listes, la première qui contient les éléments d'indice pair (L1) et la seconde les éléments d'indice impair (L2) ?
   - Exemple : L = [-5, 2, 1, 18, 0] donne L1 = [-5, 1, 0] et L2 = [2, 18]

Compréhension

Exercice : Obtenir la liste des carrés des entiers d'une liste de nombres

Solution : On sait faire ;-)

```
nombres = [4, 2, 1, 5, 6]  # la liste de nombres
carres = []
for n in nombres:
    carres.append(n ** 2)
assert carres == [16, 4, 1, 25, 36]
```

Et en math : Comment notons-nous ceci (en utilisant des ensembles) ?

- $N = \{4, 2, 1, 5, 6\}$: c'est une définition **en extension**
- $C = \{x^2 | x \in N\}$: c'est une définition **en compréhension**

Compréhension en Python :

```
carres = [ x ** 2 for x in nombres ]
assert carres == [16, 4, 1, 25, 36]
```

Intérêts de la compréhension [2] **:**

- formulation plus concise
- plus facile à utiliser car c'est une expression (voir invariant)
- MAIS attention à ne pas avoir des expressions trop compliquées

2. Le mécanisme sous-jacent, les **générateurs** a d'autres intérêts.

Compréhension (suite)

Compréhension avec filtrage : Ajout d'une condition sur les éléments à conserver.

Exemple : Les cubes des entiers pairs d'une séquence

```
nombres = [4, 2, 1, 5, 6]
cubes = tuple(x ** 3 for x in nombres if x % 2 == 0)
assert cubes == (64, 8, 216)
```

est équivalent à (les deux formulations sont très proches) :

```
cubes = []
for x in nombres:
    if x % 2 == 0:
        cubes.append(x ** 3)
cubes = tuple(cubes)
```

Remarque : Il faut écrire explicitement `tuple` sinon on obtient un générateur et non un n-uplet.

Matrices

Principe : On peut utiliser une liste de listes.

```
# Une matrice comme une liste de listes (en extension)
# Chaque liste représente une ligne de la matrice
# (ou une colonne, c'est une convention)
matrice = [[0,  1,  2,  3],
           [10, 11, 12, 13],
           [20, 21, 22, 23]]

# On remarque que dans matrice[i][j] on a i * 10 + j
assert matrice[2][1] == 21

# La même matrice en compréhension
matrice2 = [ [i * 10 + j for j in range(4) ] for i in range(3)]
assert matrice2 == matrice

# La même sans compréhension
matrice3 = []
for i in range(3):
    ligne = []
    for j in range(4):
        ligne.append(i * 10 + j)
    matrice3.append(ligne)
assert matrice3 == matrice
```

Remarque : Il existe des modules spécialisés comme Numpy.

Sequence et for

- destructuration et `for` : plusieurs noms si séquence de séquences

```
for nom, age in [('Paul', 18), ('Sarah', 19), ('Nicolas', 21), ('Emma', 16)]:
    print(nom, 'a', age, 'ans', end='.  ')
```

Affiche : 'Paul a 18 ans. Sarah a 19 ans. Nicolas a 21 ans. Emma a 16 ans.'

Exercice : Consigne
Traiter les deux exercices suivants en utilisant 1) un `while` et 2) un `for` puis comparer les deux solutions.

Exercice : équivalent de `count`
On veut calculer le nombre d'occurrences d'un élément x dans une séquence s.

Exercice : équivalent de `index`
On veut trouver l'indice de la première occurrence d'un élément x dans une séquence s.
L'indice sera None[3] si x n'est pas dans s.
On accèdera au plus une fois aux éléments de s.

3. l'opération index de Python lève une exception au lieu de retourner `None` si l'élément n'est pas dans la liste.

Solution pour équivalent de count

- Avec un `while`

```
indice: int = 0     # indice sur s.
frequence: int = 0      # nombre de x dans s[0:indice]
while indice < len(s):  # encore des éléments à considérer
    # Variant : len(s) - indice
    # Invariant : frequence = nombre d'apparition de x dans s[0:indice]
    if s[indice] == x:
        frequence += 1
    indice += 1
```

- Avec un `for`

```
frequence: int = 0      # nombre de x trouvés dans s
for elt in s:
    if elt == x:
        frequence += 1
```

- **Constat :** La version avec `for` est plus naturelle
- **Compréhension :**

```
frequence = sum(1 for elt in s if elt == x)
```

Solution pour équivalent de index

- Avec un `while`

```
indice: int = 0    # indice sur s.
while indice < len(s) and s[indice] != x:  # encore des éléments ET pas le bon
    # Variant : len(s) - indice
    # Invariant : x not in s[0:indice]
    indice += 1
if indice >= len(s): # x n'a pas été trouvé
    indice = None
```

- Avec un `for`

```
indice: int = 0    # indice sur s.
for elt in s:
    # Invariant : x not in s[0:indice]
    if elt == x:
        break
    indice += 1
else:
    indice = None
```

- **Constats :**
  - on ne peut pas préciser de condition de sortie dans un `for` d'où l'utilisation de `break`
  - le bloc `else` ne sera exécuté que si aucun `break` n'a été exécuté (et donc x non trouvé)
  - la gestion de l'indice alourdit le `for` !

enumerate et zip

- **enumerate** est une fonction qui « retourne » [4] autant de couples qu'il y a d'éléments dans une séquence. Chaque couple est composé d'un numéro d'ordre et d'un élément de la séquence.

```
assert list(enumerate('ABC')) == [(0, 'A'), (1, 'B'), (2, 'C')]
```

- Nouvelle solution pour index avec un `for` et enumerate : simplifie le code !

```
for indice, elt in enumerate(s):
    if elt == x:
        break
else:
    indice = None
```

- **zip** : « retourne » [5] les couples formés en prenant successivement un élément de chacune des séquences fournies en paramètre.
- la plus petite séquence détermine le nombre de couples

```
assert list(zip(range(1, 4), 'ABCD',  [3, 2, 5, 7])) \
       == [(1, 'A', 3), (2, 'B', 2), (3, 'C', 5)]
```

4. C'est en fait un générateur.
5. C'est en fait un générateur.

Sommaire

Principe

Motivation

Comment faire pour construire un programme non trivial (quelques dizaines de lignes ou plus) ?

Principe de la méthode des raffinages

- Le programme est vu comme une action complexe
- Cette action est décomposée en sous-actions combinées grâce aux structures de contrôle
- Les sous-actions sont à leur tour décomposées jusqu'à obtenir des actions élémentaires.

C'est une approche classique !

Document, rapport, article, etc.
- plan, souvent explicité par la table des matières
  au moins annoncé dans l'introduction !

Recette de cuisine
- décrit les étapes à suivre
- des étapes complexes (décrites en début d'ouvrage) :
  - abaisser la pâte
  - préparer une pâte brisée...

Mode d'emploi
- description des actions à réaliser par l'utilisateur sous forme de listes numérotées

Mathématiques
- Recours à des lemmes pour démontrer un théorème

Les étapes

Principales étapes

1. Comprendre le problème
2. Trouver une solution informelle
3. Structure cette solution (raffinages)
4. Produire le programme correspondant
5. Tester le programme

Exemple fil rouge

Afficher le pgcd de deux entiers strictement positifs.

Comprendre le problème

Motivation

Il est essentiel de bien comprendre le problème posé pour avoir des chances d'écrire le **bon** programme !

Moyens

1. Reformuler le problème en rédigeant R0
   - R0 est l'action complexe qui synthétise le travail à faire
2. Lister des « exemples d'utilisation » du programme. Il s'agit de préciser :
   - les données en entrées
   - **et** les résultats attendus

Intérêt des exemples

- Les exemples sont un moyen de spécifier ce qui est attendu
- Ils donneront les tests fonctionnels qui permettront de tester le programme

Comprendre le problème posé : exemple sur le pgcd

Reformulation

R0 : Afficher le pgcd de deux entiers strictement positifs

Remarque : La reformulation doit être concise et précise, au risque d'être incomplète.
Il s'agit de synthétiser le problème posé.

Exemples

```
   a     b    ==> pgcd
-------------------------------------------------------------
   2     4    ==>  2                    -- cas nominal (a < b)
  20    15    ==>  5                    -- cas nominal (a > b)
  20    20    ==>  20                   -- cas limite  (a = b)
  20     1    ==>  1                    -- cas limite
   1     1    ==>  1                    -- cas limite
   0     4    ==>  Erreur : a <= 0      -- cas d'erreur (robustesse) : resaisie
   4    -4    ==>  Erreur : b <= 0      -- cas d'erreur (robustesse) : resaisie
-------------------------------------------------------------
```

Trouver une solution informelle

Objectif

Identifier une manière de résoudre le problème.

Moyen

- Il s'agit d'avoir l'idée, l'intuition de comment traiter le problème.
- Comment trouver l'idée ? **C'est le point difficile !**
- On peut s'appuyer sur son expérience, son imagination, des résultats existants, etc.

Exemple du pgcd

Pour calculer le pgcd d'un nombre, on peut appliquer l'algorithme d'Euclide :

> *Il s'agit de soustraire au plus grand nombre le plus petit. Quand les deux nombres sont égaux, ils correspondent au pgcd des deux nombres initiaux.*

Remarque

On peut vérifier la solution informelle sur les exemples d'utilisation identifiés.

Exemple : 20 et 15 $\rightarrow$ 5 et 15 $\rightarrow$ 5 et 10 $\rightarrow$ 5 et 5 $\rightarrow$ 5.

Structurer la solution

Objectif

- Formaliser la solution informelle
- Décomposer une action complexe en sous-actions
- Identifier les structures de contrôle qui permettent de combiner les sous-actions
- On répond à la question **Comment ?**

Notation

On note **R1** la décomposition, le raffinage, de R0.

Définition

Un raffinage est la décomposition d'une action complexe A en une combinaison d'actions (les sous-actions) qui réalise exactement le même objectif que A.

Exemple sur le pgcd

```
R1 : Comment « Afficher le pgcd de deux entiers positifs » ?
    Saisir deux entiers
    { les deux entiers sont strictements positifs }
    Déterminer le pgcd des deux entiers
    Afficher le pgcd
```

Xavier Crégut <prenom.nom@enseeiht.fr>

Raffinages et flot de données

Objectif

- Expliciter les données manipulées par une sous-actions d'un raffinage
- Préciser ce que fait la sous-action de la donnée :
  - **in** : elle l'utilise sans la modifier
  - **out** : elle produit cette donnée sans consulter sa valeur
  - **in out** : elle l'utilise, puis la modifie

Exemple du pgcd

```
R1 : Comment « Afficher le pgcd de deux entiers positifs »
    Saisir deux entiers a et b      a, b: out
    { (a > 0) Et (b > 0) }
    Déterminer le pgcd de a et b    a, b: in; pgcd: out
    Afficher le pgcd                pgcd: in
```

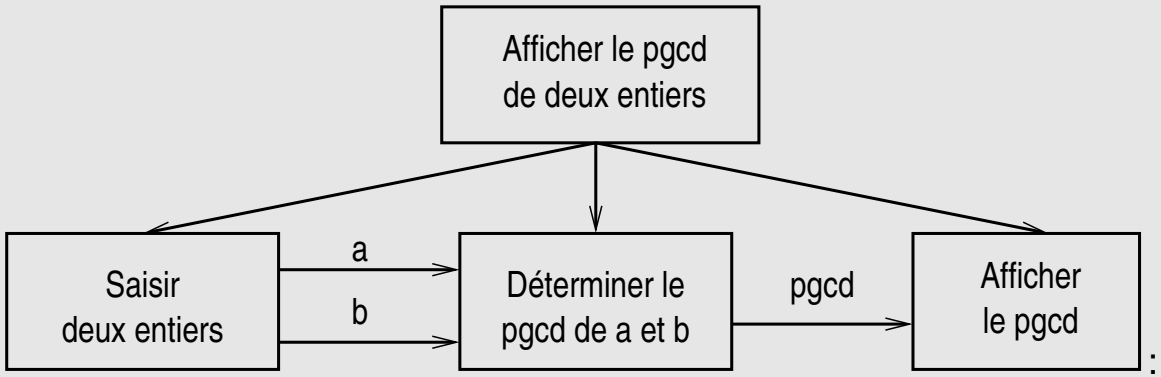
Avantages

Expliciter les données permet de :

- les référencer dans les autres actions
- d'écrire plus formellement les propriétés du programme (entre accolades)
- de contrôler le séquencement des actions :
  - une donnée doit être produite (**out**) avant d'être utilisée (**in**)

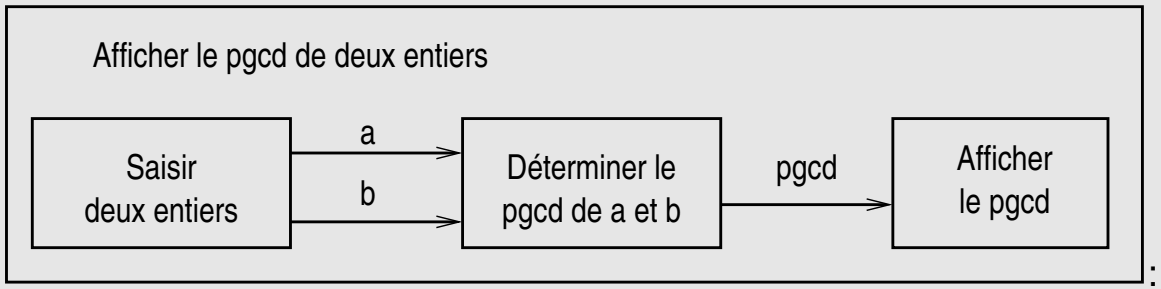
Représentation graphique

Sous forme d'arbre



:

Sous forme de boîtes imbriquées



:

Remarques

- Permet de visualiser la signification de **in** et **out** (le point de vue est l'action).
- Peu adapté si combinaison complexe des sous-actions.

Continuer à raffiner

Principe

Si un raffinage introduit des actions complexes, elles doivent être à leur tour raffinées.

Les 3 actions introduites dans la décomposition de R0 sont complexes et doivent donc être raffinées.

Notation

- On note R2 tout raffinage d'une action introduite dans R1.
- Plus généralement, on note R_i le raffinage d'une action introduite en R_{i-1}.
- Attention, il faut préciser l'action qui est décomposée.

```
Ri: Comment « Action complexe » ?
␣␣␣␣Combinaison d'actions réalisant « Action complexe »
```

Raffinages (définition)

On appelle **raffinages** l'arbre dont R0 est la racine, les feuilles les actions élémentaires et les nœuds les actions complexes identifiées. On appelle **raffinage** la décomposition d'une action complexe.

Remarques

- On peut arrêter de raffiner si une action est déjà connue (élémentaire ou déjà raffinée)
- On peut ne pas décomposer une action complexe si elle est simple ou immédiate (subjectif)
- À chaque nouvelle décomposition, on peut se poser la question de la solution informelle

Continuer le raffinage du pgcd

Raffinages de niveau 2

```
R2 : Comment « Déterminer le pgcd a et b » ?
    na = a    -- variables auxiliaires car a et b sont en in
    nb = b    -- et ne devraient donc pas être modifiées.
    while na et nb différents:                    na, nb: in
        Soustraire au plus grand le plus petit    na, nb: in out
    pgcd = na   -- pgcd était en out, il doit être initialisé.


R2 : Comment « Afficher le pgcd » ?
    print("pgcd =", pgcd)


R2 : Comment « Saisir deux entiers » ?
        ...
```

- Bien sûr, un raffinage peut utiliser des structures de contrôle pour combiner les sous-actions !
- Ici, nous utilisons celle de Python pour éviter de multiplier les langages.
- Le raffinage d'une action complexe est donc un programme Python :
  - qui utilise des structures de contrôle (séquence, conditionnelles, répétitions)
  - pour combiner des actions élémentaires ou complexes

Continuer le raffinage du pgcd

Raffinages de niveau 3

```
R3 : Comment évaluer « na et nb différents » ?
    na != nb


R3 : Comment « Soustraire au plus grand le plus petit » ?
    if na > nb:
        na = na - nb
    else:
        nb = nb - na
```

Raffinage d'une expression

- Le premier R3 ne raffine pas une action mais une expression complexe.
- Il s'agit donc d'expliquer comment on obtient son résultat
- On l'écrit directement.
- On pourrait initialiser une variable `résultat` dont la valeur sera la valeur de l'expression

```
R3 : Comment évaluer « na et nb différents » ?
    résultat =  na != nb
```

Qualités d'un raffinage

Présentation des raffinages

1. Les raffinages commencent par R0 et des exemples (R0 est une action)
2. Un raffinage doit être bien présenté (indentation).

```
Ri : Comment « action complexe » ?
     actions combinées au moyen de structures de contrôle
```

3. Tous les Ri sont écrits contre la marge.
4. Un raffinage ne doit pas apparaître avant l'action complexe qu'il décompose
5. Une action complexe commence par un verbe à l'infinitif

Règles

1. Le vocabulaire utilisé doit être précis et clair
2. Chaque niveau de raffinage doit apporter suffisamment d'information (mais pas trop).
   - Il faut trouver le bon équilibre !
3. Le raffinage d'une action (combinaison des sous-actions) doit décrire totalement cette action
4. Le raffinage d'une action ne doit décrire que cette action
5. Éviter les structures de contrôle déguisées (si, tant que) : les expliciter (`if`, `while`...)
6. Ne pas utiliser « Comparer », « Vérifier »... et préférer « Déterminer », « Calculer »...

- Certaines de ces règles sont subjectives !
- L'important est de pouvoir expliquer et justifier les choix faits

Vérification d'un raffinage

Indices d'actions complexes manquantes

1. Plusieurs conditionnelles ou répétitions dans un même raffinage
2. Beaucoup d'actions dans le raffinage (par exemple plus de 5 ou 6)

Liens entre action complexe (C) et les sous-actions (A) de son raffinage

1. La combinaison d'actions du raffinage de C est la réponse à la question "Comment « C » ?"
   - par définition !
2. Si la réponse à « Pourquoi A ? » n'est pas C, alors :
   - soit on a trouvé un meilleur nom pour A (ou C)
   - soit on a identifié une action complexe intermédiaire entre C et A (il faut l'ajouter)
   - soit A n'est pas à sa place (ne fait pas partie des objectifs de C)

Flots de données

Utiliser les flots de données :

- pour vérifier les communications entre niveaux :
  - les sous-actions doivent produire les résultats (**out**) de l'action ;
  - les sous-actions peuvent (doivent) utiliser les entrées de l'action.
- l'enchaînement des actions au sein d'un niveau :
  - une donnée ne peut être utilisée (**in**) que si elle a été produite (**out**) avant

Produire l'algorithme

Dictionnaire des données

La liste des données (variables) utilisées avec leur signification.

Algorithme

Un *algorithme* est la mise à plat des raffinages :

- R0 devient le commentaire général de l'algorithme.
- En faisant un parcours préfixe (le nœud avant les fils),
  - les actions élémentaires sont les instructions ;
  - les actions complexes deviennent des commentaires.

Remarque

L'obtention de l'algorithme est direct si les derniers niveaux des raffinages ne contiennent pas d'actions complexes sinon il faut décomposer ces actions complexes !

Programme

C'est la traduction de l'algorithme dans un langage de programmation.
Ici les deux coïncident car nous avons utilisé les structures de contrôle Python dans les raffinages

Programme du pgcd en Python

afficher_pgcd.py

```
1  def afficher_pgcd():
2      '''Afficher le pgcd de deux entiers strictement positifs.'''
3
4      # Saisir deux entiers a et b
5      ...
6
7      assert a > 0
8      assert b > 0
9
10     # Déterminer le pgcd de a et b
11     na = a  # nouveau a pour ne pas modifier a
12     nb = b  # idem pour b
13     while na != nb: # na et nb différents
14         # Soustraire au plus grand le plus petit
15         if na > nb:
16             na = na - nb
17         else:
18             nb = nb - na
19     pgcd = na   # le pgcd de a et b
20
21     # Afficher le pgcd
22     print('pgcd =', pgcd)
```

Xavier Crégut <prenom.nom@enseeiht.fr>

Tester le programme

Tester

processus d'exécution d'un programme avec l'intention de découvrir des erreurs !

Grands types de test

Fonctionnels : Le programme fait-il ce qu'on veut qu'il fasse ? (programme = « boîte noire »)
Structurels : Toutes les parties du code ont-elles été exercées ? (programme = « boîte blanche »)

Échantillonnage

Repose sur l'hypothèse implicite que si le programme testé fournit des résultats corrects pour l'échantillon choisi, il fournira aussi des résultats corrects pour toutes les autres données.

Comment choisir l'échantillon ?

- Tests fonctionnels : on peut établir une matrice pour vérifier que chaque exigence a été testée
- Tests structurels : on peut s'appuyer sur la notion de taux de couverture :
  - A-t-on exécuté au moins une fois toutes les instructions ?
  - Est-on passé au moins une fois par toutes les décisions (branchement) ?...

Problème de l'oracle

Comment savoir que le résultat du programme est correct ?

On connait déjà des algorithmes : multiplication de deux entiers

Trois techniques pour calculer 124 * 735 :

```
Technique 1          |      Technique 2          |          Technique 3
                     |                           |
     124             |           1  2  4         |     124    735       124
   * 735             |       *   7  3  5         |     248    367       248
   -----             |       ----------          |     496    183       496
     620             |          05 10 20         |     992     91       992
    372              |       03 06 12            |    1984     45      1984
   868               |    07 14 28               |    3968     22
   -----             |    --------------         |    7936     11      7936
   91140             |    07 17 39 22 20         |   15872      5     15872
                     |    --------------         |   31744      2
                     |     9  1  1  4  0         |   63488      1     63488
                     |                           |                    -----
                     |                           |                    91140
```

Comment marchent-elles ?

Exercices

Exercice : Expliquer les techniques de multiplications

Pour chacune des techniques de calcul de la multiplication présentées sur la planche précédente :

1. Indiquer les opérations élémentaires sur lesquelles s'appuient la technique.
2. Utiliser la méthode des raffinages pour expliquer la technique.

Exercice : Parenthéser une expression

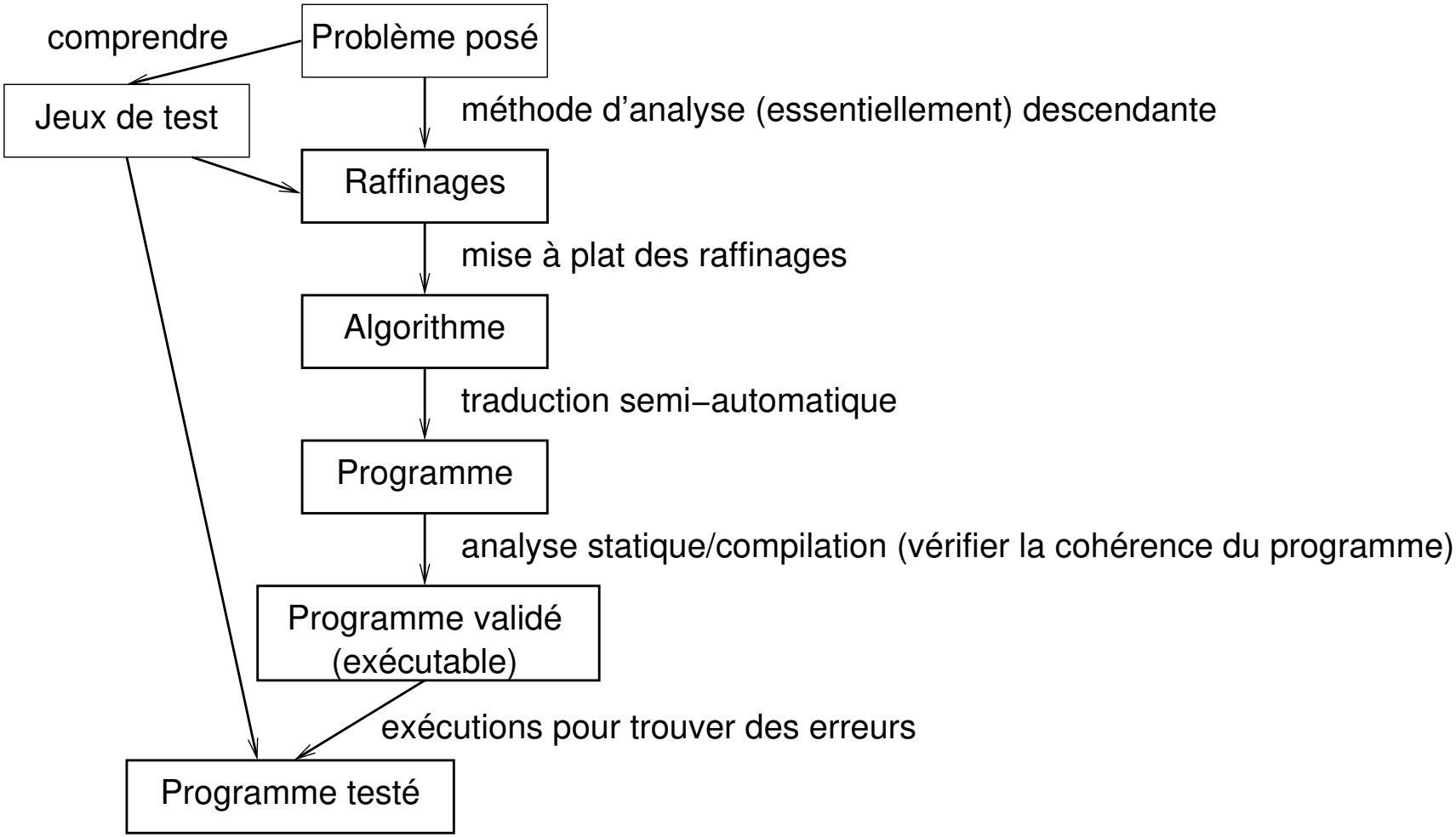
Utiliser la méthode des raffinages pour expliquer comment parenthéser une expression.

Exemple : `5 * x + 3 * y ** 2` est équivalente à `((5 * x) + (3 * (y ** 2)))`

Exercice : Comment construire un algorithme

Utiliser la méthode des raffinages pour expliquer comment construire un algorithme en utilisant la méthode des raffinages.

Développement d'un programme



Sommaire

Motivation

Objectif

Le but des sous-programmes est de permettre au programmeur de définir ses propres **instructions (procédures)** ou ses propres **expressions (fonctions)** sous la forme de **sous-programmes**.

Les sous-programmes permettent d'enrichir le langage de nouvelles instructions/expressions.

Fonction et Procédure

Une **fonction** est un sous-programme qui retourne un résultat (une valeur).

Exemples : abs, int, float, randint, etc.

Une **procédure** est un sous-programme qui ne retourne pas de résultat.

Exemple : print

En Python **tout est fonction** : Une procédure est une fonction qui retourne toujours **None**.

Identification des sous-programmes

Toute action ou expression complexe identifiée dans un raffinage est un sous-programme potentiel.

Introduction

Question

Quelle est la solution de l'équation $2x + 3 = 0$?

Réponse

La solution est $-3/2 = -1,5$

Questions

Quelles sont les solutions des équations suivantes ?

- $2x + 4 = 0$
- $2x + 8 = 0$
- $8x + 8 = 0$
- $5x - 8 = 0$

Généralisons

- On peut généraliser le problème : on veut connaître la solution de $ax + b = 0$,
- a et b sont les données du problème.
- La solution est alors $x = -b/a$
- Est-ce toujours la solution ?

Fonction, paramètres formels et paramètres effectifs

```python
#! Définition d'une fonction
def solution_affine(a, b):
    return -b / a
```

```python
#! Utilisation de la fonction
x1 = solution_affine(2, 3)  ; assert x1 == -1.5
x2 = solution_affine(2, 4)  ; assert x2 == -2.0
x3 = solution_affine(5, -8) ; assert x3 == 1.6
```

Définition de la fonction `solution_affine`

- `solution_affine` est notre première **fonction** (mot-clé `def`)
- *a* et *b* sont des **paramètres formels**
  - on ne sait pas quelle est leur valeur
  - mais quand la fonction sera exécutée cette valeur sera connue
  - on peut donc raisonner dessus
- Un bloc doit suivre la ligne `def` : ce bloc sera exécuté lors de l'appel de la fonction
- L'instruction « `return expr` » termine l'exécution de la fonction :
  - `expr` est une expresssion dont la valeur est le résultat de la fonction
  - ici, le résultat est `-b / a`

Utilisation de la fonction `solution_affine`

- les paramètres formels sont initialisés lors de l'appel de la fonction :
  - 2 et 3 ou 2 et 4 ou 5 et -8 sont les **paramètres effectifs** (ou **paramètres réels**)
  - ils servent à initialiser *a* et *b*
  - le ième paramètre effectif initialise le ième paramètre formel
  - dans l'appel `solution_affine(2, 3)` la fonction est exécutée avec `a == 2` et `b == 3`
  - on parle de paramètres *positionnels* (identifiés par leur position)
- **autre appel possible :** `solution_affine(b=3, a=2)`
  - on lie explicitement le paramètre effectif au paramètre formel (avec `nom_formel=effectif`)
  - on peut initialiser les paramètres dans n'importe quel ordre (*paramètres nommés*)

Analogie avec les fonctions en math

Exemple de fonction en math

- $f : x \mapsto x^2 + 5x + 6$
- $f(x) = x^2 + 5x + 6$
- x est une variable
- $x^2 + 5x + 6$ est l'image par f de x
- Utilisation :
  - $f(2) = 2^2 + 5.2 + 6 = 20$
  - $f(-3) = (-3)^2 + 5.(-3) + 6 = 9 - 15 + 6 = 0$

La même fonction en Python

```
def f(x):
    return x ** 2 + 5 * x + 6

y1 = f(2)
assert y1 == 20
y2 = f(-3)
assert y2 == 0
assert f(-2) == 0
```

Règles sur les paramètres lors de l'appel d'une fonction

- C'est **seulement** l'appel du sous-programme qui lie paramètres formels et paramètres effectifs
  - soit par leur position (**paramètres positionnels**) f(1, 2)
  - soit par l'association nom_formel=effectif (**paramètres nommés**) f(a=1, b=2)
- Python est plus riche : valeur par défaut, paramètres obligatoirement nommés, nombre variable de paramètres, paramètres obligatoirement positionnels. . .

Règles

- Tous les paramètres formels doivent recevoir une et une seule valeur
- Quand on nomme un paramètre effectif, on doit nommer les suivants

Exemple

```
def f(a, b):
    print(a, b)

f(1, 2)        # a est lié à 1 et b à 2 (affiche : 1 2)
f(a=1, b=2)    # Affiche : 1 2
f(b=2, a=1)    # Affiche : 1 2
f(1, b=2)      # Affiche : 1 2
f(b=2)         # TypeError: f() missing 1 required positional argument: 'a'
f(a=1, 2)      # SyntaxError: positional argument follows keyword argument
f(1, 2, a=1)   # TypeError: f() got multiple values for argument 'a'
```

Tester une fonction

Fonction de test

Une bonne pratique consiste à définir une fonction qui regroupe les exemples faits/à faire.
Par convention, son nom commence par test. C'est une **fonction de test**.

Exemple

```
def test_affine():
    assert solution_affine(2, 3)  == -1.5
    assert solution_affine(2, 4)  == -2.0
    assert solution_affine(5, -8) == 1.6
```

Utilisation

- Appeler la fonction de test pour jouer les exemples (tests) : `test_affine()`
- Si des erreurs sont signalées, il y a un problème dans la fonction ou le test
- **Conseil :** Écrire la fonction de test avant le code de la fonction :
  - écrire les tests/exemples, c'est vérifier qu'on a compris ce qui est demandé
  - exécuter les tests sans le code de la fonction écrit permet de voir qu'ils détectent des erreurs
  - voir qu'il n'y a plus d'erreurs une fois le code de la fonction écrit donne confiance

Tester automatiquement

Il existe des outils comme pytest pour exécuter automatiquement ces méthodes de test.

Xavier Crégut <prenom.nom@enseeiht.fr>

Documentation

Motivation

Une fonction peut être utilisée dans différents contextes. Il est donc important, pour tous ceux qui vont l'utiliser, de bien comprendre comment elle se comporte.

Moyen : Chaîne de documentation

La définition de la fonction **doit** commencer une chaîne de documentation (docstring) qui contient :

- une phrase pour décrire l'objectif, suivie d'une ligne blanche
- une description plus détaillée (conditions d'utilisation, effets)
- une description des paramètres, du résultat, etc.

La fonction `help` de Python exploite cette chaîne. Essayer `help(solution_affine)`.

Nouvelle définition de solution_affine (plusieurs styles possibles)

```
def solution_affine(a, b):
    '''Retourne la solution de l'équation a * x + b == 0.

    :param a: le coefficient de x
    :param b: le terme constant
    :returns: un réel solution de l'équation affine.
    '''
    return - b / a
```

Question : A-t-on bien explicité les conditions d'utilisation ?

Exercice : Cube d'un nombre

Écrire une fonction qui calcule le cube d'un nombre et son programme de test.

Exemples : le cube de 3 est 27, le cube de -0.5 est -0.125

Exercice : Somme de *n* premiers entiers

Écrire une fonction qui calcule la somme des *n* premiers entiers et son programme de test.

Exemples : pour n vaut 5, on obtient 15, pour n vaut 3, on obtient 6.

Exercice : Appels de fonctions

Indiquer si les appels suivants sont valides ou non. Si invalides, expliquer pourquoi.

```
1  def f(a, b, c):
2      pass
3
4  f(1, 2, 3)
5  f(1)
6  f(c=1, a=1, b=1)
7  f(a=b=c=1)
8  f(1, c=3, b=2)
9  f(1, b=2, 3)
10 f(1, 3, b=2)
11 f(1, 2, c=3)
12 f(a=1, 2, 3)
```

Anatomie d'une fonction

Une fonction est définie grâce au mot-clé `def` et comporte deux parties :

1. la **spécification** qui décrit ce que fait la fonction (le QUOI), suffisante pour les utilisateurs
2. L'**implantation** : le code qui réalise cette spécification (le COMMENT)

La **spécification** d'une fonction est composée de :

- une **signature** : la ligne qui contient `def`, le nom de la fonction, ses paramètres formels
- une **sémantique** : une chaîne de caractères (docstring) qui décrit l'objectif de la fonction
- la signature est suffisante pour Python ; la sémantique est nécessaire pour l'utilisateur

La **signature** respecte la syntaxe suivante : `def nom_fonction(paramètre1, ...):`

- où apparaissent le nom de la fonction et de ses paramètres formels
- les « deux-points » en fin de ligne ouvrent sur un bloc (les instructions de la fonction)
- attention à indenter les lignes qui suivent ces « deux-points » !
- la première ligne du bloc est la chaîne de documentation (docstring) pour la sémantique

La **sémantique** (docstring) est normalisée : voir PEP 257 et différents styles

L'**implantation** ou **corps** qui suit la chaîne de documentation (docstring)

- ce sont les instructions déjà vues !
- elles sont exécutées quand la fonction est appelée
- `return` arrête l'exécution de la fonction et indique la valeur retournée
- comment obtenir l'implantation ? Appliquer la méthode des raffinages avec R0 = spécification

En général, on définit plusieurs fonctions dans le même fichier (voir Modules)

Typage

Motivation

Il est utile de pouvoir expliciter le type attendu d'un paramètre formel pour vérifier, avant l'exécution, que le type du paramètre effectif correspondant est compatible.
Idem pour le type de retour d'une fonction.

Solution : PEP 484 : Type Hints

```
def solution_affine(a: float, b: float) -> float:
    ''' ... '''
    return - b / a
                                #### Y a-t-il des erreurs dans la suite ? ####
from datetime import datetime
if datetime.today().day == 31:    # condition rarement vraie !
    u: float = 6.4 ; v: int = -2    # À éviter : mettre une seule instruction par ligne !
    s: float = solution_affine(v, u)
    t: int = solution_affine(v, u)
    x = solution_affine('5.0', 3)
```

Exploitation : les types sont ignorés par Python mais utilisés par mypy

```
> python affine.py       # pas d'erreur...
    # ...sauf le 31 du mois : TypeError: unsupported operand type(s) for /: 'int' and 'str'
> mypy affine.py         # outil qui vérifie le typage statique optionnel
affine.py:10: error: Incompatible types in assignment (expression has type "float", variable has type "int")
affine.py:12: error: Argument 1 to "solution_affine" has incompatible type "str"; expected "float"
```

Problème

Questions

1. Quelles sont les solutions de l'équation $a * x + b = 0$ quand $a = 0$?
2. Quelles sont les conséquences sur la fonction `solution_affine` ?

Solutions de $a * x + b = 0$ si $a = 0$

- Aucune solution si $b \neq 0$. C'est donc une **fonction partielle**.
- Tout réel est solution si $b = 0$. Ce n'est pas une fonction au sens mathématiques.

Conséquences

- La fonction que l'on a écrite provoquera une division par zéro si $a = 0$.
- Il faut donc la corriger !
- Deux solutions :
  1. programmation par contrat : l'appelante et l'appelée se font confiance et décident de qui fait quoi
  2. programmation défensive : pas de confiance. L'appelée doit envisager tous les cas.

Remarques

- Nous aurions dû identifier ce problème plus tôt
- Dès qu'on écrit une instruction ou une expression, il faut s'assurer qu'elle a un sens
- En particulier `- b / a` n'a de sens que si la valeur de `a` est non nulle

Xavier Crégut <prenom.nom@enseeiht.fr>

Version en programmation par contrat

Principe :

- **spécifier qui fait quoi** au moyen de préconditions (**pre**) et postconditions (**post**) sur la fonction
- **Préconditions :** conditions sur les paramètres en entrée de la fonction
- **Postconditions :** conditions sur les paramètres en sortie (résultats) de la fonction
  - on suppose qu'il existe une variable **result** qui correspond au résultat de la fonction
- Préconditions et postconditions définissent le **contrat** de la fonction :
  - L'appelante doit vérifier que les préconditions sont satisfaites avant l'appel de la fonction
  - L'appelée doit garantir les postconditions après l'exécution (si les préconditions étaient satisfaites)
- On peut instrumenter le **contrat** (préconditions et postconditions) en utilisant `assert`

```
def solution_affine(a: float, b: float) -> float:
    '''
    Retourne la solution de l'équation a * x + b == 0.

    :param a: le coefficient de x
    :param b: le terme constant
    :pre: a != 0,   'a non nul'
    :returns: un réel solution de l'équation affine.
    :post: a * result + b == 0, 'solution' #! il faudrait le vérifier à epsilon près'
    '''
    assert a != 0                          #! instrumentation de la précondition
    result = - b / a                       #! variable nécessaire pour exprimer la post
    assert abs(a * result + b) <= 1e-8     #! instrumentation de la postcondition
    return result
```

Xavier Crégut <prenom.nom@enseeiht.fr>

Programmation par contrat : Utilisation de la fonction

Exemple de programme qui utilise la fonction `solution_affine` :

- dans `tester`, on sait que a est non nul, donc pas de test explicite
- dans `resoudre_affine`, on s'assure que l'utilisateur fournit un a non nul (**`while`**)

```
def tester():
    '''Tester avec des valeurs prédéfinies'''
    tests = ((2, -4, 2), (4, -3, .75), (1.5, 4.5, -3.0)) # ((a, b, solution)...)
    for a, b, attendu in tests:
        assert a != 0.0    #! On sait que a est non nul, cf tests ci-avant
        assert attendu == solution_affine(a, b), f'pour {a}*x + {b} = 0 : ' \
               f'{solution_affine(a, b)} au lieu de {attendu}'

def resoudre_affine():   #! Lignes blanches supprimées pour gain de place
    '''Afficher la solution d'une équation affine pour a et b saisis...'''
    # Demander à l'utilisateur la valeur de a (!= 0)
    a = float(input('a = '))       # demander une valeur pour a
    while a == 0.0:                # Valeur de a incorrecte
        print('a doit être != 0') # signaler l'erreur
        a = float(input('a = '))  # demander une nouvelle valeur pour a
    assert a != 0.0
    # Demander à l'utilisateur la valeur de b
    b = float(input('b = '))
    # Calculer la solution
    solution = solution_affine(a, b)  #! On sait que a != 0.0 (saisie contrôlée)
    # Afficher la solution
    print(f'La solution de {a}*x + {b} = 0 est {solution}.')
```

Programmation par contrat : Bilan

Prérequis

- Il faut que l'appelée puisse faire confiance à l'appelant (il respectera les préconditions)

Intérêts

1. **définition des responsabilités :** chacun sait qui fait quoi
   - en cas d'erreur détectée, on connait le responsable : appelée (pré) ou appelant (post)
2. **documentation :** les fonctions sont documentés formellement, sans ambiguïtés
3. **aide à la mise au point** si elles sont instrumentées et vérifiées pendant l'exécution (`assert`...)
   - les erreurs sont détectées au plus près de leur origine
   - elles sont donc plus facile à localiser et corriger
4. **exploitable par outils d'analyse statique** et **générateurs de tests**
5. **code final optimisé** (pas d'instrumentation) : seuls les tests nécessaires sont faits.

Conclusion

- C'est la solution que nous allons privilégier pour l'instant

Difficultés

- Il n'est pas toujours facile d'exprimer les contrats (en particulier les postconditions)
- Donner le contrat d'une opération qui ajoute un élément x en position i dans une séquence s.

Version en programmation défensive

Principe : Rendre la fonction totale

- Les cas hors limites doivent être gérés (la précondition doit être True)
- On choisit une valeur particulière pour les cas où la fonction n'est pas définie
  - `None` quand il n'y a pas de solution
  - `'R'` quand tout réel est solution de l'équation
  - Le **mécanisme d'exception** serait plus adapté (la fonction reste partielle)
- Le type de retour se complique : Optional (car peut être None) et Union (car float ou str) !

```python
from typing import Union, Optional

def solution_affine(a: float, b: float) -> Optional[Union[float, str]]:
    '''Retourne les solutions de l'équation a * x + b == 0.

    :param a: le coefficient de x
    :param b: le terme constant
    :returns:
        * le réel solution de l'équation affine si a != 0,
        * 'R' si a et b sont nuls,
        * None dans les autres cas'''
    if a != 0:
        return - b / a
    elif b == 0:
        return 'R'
    else:
        return None

# exemple d'utilisation
assert solution_affine(2, -6) == 3
assert solution_affine(2, -7) == 3.5
assert solution_affine(0,  0) == 'R'
assert solution_affine(0, 10) == None
```

Règles sur l'exécution d'une fonction

Pour comprendre l'appel d'une fonction de la forme : `v = f(a1, ..., an)`, il faut savoir comment s'exécute l'appel de la fonction et l'instruction `return`. Le reste est déjà connu.

L'**intepréteur Python vérifie** que :

- la fonction existe : le nom doit correspondre à une fonction (en fait un *callable*)
- les paramètres effectifs donnent une valeur à tous les paramètres formels de la fonction

Le **programmeur doit s'assurer que** :

- les types des paramètres effectifs sont compatibles avec les types des paramètres formels
- les préconditions de la fonction sont satisfaites.

L'**appel de la fonction** correspond alors à :

1. évaluer les paramètres effectifs
2. créer un bloc d'activation dans la pile d'exécution pour :
   - conserver la ligne de l'appel
   - définir un espace de nom pour les paramètres et variables locales de la fonction
3. initialiser les paramètres formels à partir des paramètres effectifs
4. définir la prochaine instruction à exécuter : la première instruction de la fonction

L'**exécution d'un `return`** (ou de la fin de la fonction) :

1. La valeur de l'expression après `return` est calculée : R
2. Le point d'appel de la fonction est retrouvé (dans la pile d'exécution)
3. L'exécution de l'instruction contenant l'appel se poursuit sachant que l'appel prend la valeur R

Remarque : En l'absence de `return`, Python ajoute un `return None` implicite en fin de la fonction.

Variété des paramètres

Considérons les appels suivants :

```
len("bonjour")      # 1 paramètre positionnel
print(421)          # 1 paramètre positionnel
print('n =', n)     # mais il peut y en avoir un nombre quelconque
print(a, b, sep=' ; ', end='.\n')   # 2 paramètes positionnels
            # et deux paramètres nommés (end et sep)
```

Question : Plusieurs sous-programmes nommés `print` ?
Non, pas de surchage en Python. Il n'y qu'un seul `print` !

Vocabulaire :

- paramètre formel (parameter) : lors de la définition du sous-programme
  - exemple : def len(obj): ==> obj est un paramètre formel
- paramètre effectif ou réel (argument) : lors de l'appel du sous-programme
  - exemple : len("bonjour") ==> obj (formel) référence "bonjour" (effectif)
- paramètre positionnel : l'association entre effectif et formel se fait grâce à la position
- paramètre nommé : l'association entre effectif et formel se fait par le nom du paramètre formel
- nombre variable de paramètres effectifs : nombre de paramètres effectifs non connu lors de la spécification du SP
  - exemple : print, max, min, etc.

Paramètres positionnels

Paramètres positionnels

Un paramètre positionnel est identifié par sa position.
Le ième **paramètre effectif** correspond au ieme **paramètre formel**.

```
def f(a, b):
    print(a, b)
f(1, 2)      # a est lié à 1 et b à 2 (affiche : 1 2)
```

Appel en nommant les paramètres

Lors de l'appel, on peut nommer les paramètres.

```
f(a=1, b=2)    # Affiche : 1 2
f(b=2, a=1)    # Affiche : 1 2
f(1, b=2)      # Affiche : 1 2
f(b=2)         # TypeError: f() missing 1 required positional argument: 'a'
f(a=1, 2)      # SyntaxError: positional argument follows keyword argument
f(1, 2, a=1)   # TypeError: f() got multiple values for argument 'a'
```

Règles

- Tous les paramètres formels doivent recevoir une et une seule valeur
- Quand on nomme un paramètre effectif, on doit nommer les suivants

Valeur par défaut d'un paramètre

Règle : valeur par défaut

- On peut donner une valeur par défaut à un paramètre formel positionnel.
- Il faut alors donner une valeur par défaut à tous les paramètres positionnels suivants.
- Lors de l'appel, si on omet un paramètre effectif, sa valeur par défaut sera utilisée.

Exemple

```
def f(a, b=2, c=3):
    print(a, b, c)
f(1, 0, 9)     # 1 0 9
f(1)           # 1 2 3
f(1, 0)        # 1 0 3
f(1, c=9)      # 1 2 9
```

Danger : les valeurs par défaut sont évaluées lors de la définition de la fonction

```
def g(x, s = []):    # [] évalué une fois à la définition de la fonction
    s.append(x)
    return s
g(1)    # [1]
g(2)    # [1, 2]    # On n'a donc pas une nouvelle liste vide à chaque appel à g
```

Comment faire pour avoir une nouvelle liste à chaque fois ?

Nombre variable de paramètres positionnels

Motivation

Pouvoir appeler une fonction avec un nombre variable de paramètres effectifs.

```
print(1)
print(1, 2, 3)
```

Principe

Faire précéder par * le dernier paramètre positionnel qui sera alors un n-uplet contenant tous les paramètres effectifs en surnombre (non associés à un paramètre formel positionnel).

Exemple

```
def f(a, b=9, *c):
    print('a={}, b={}, c={}'.format(a, b, c))
f(1)                # a=1, b=9, c=()
f(1, 4)             # a=1, b=4, c=()
f(1, 2, 3)          # a=1, b=2, c=(3,)
f(1, 2, 3, 4, 5)    # a=1, b=2, c=(3, 4, 5)
f(1, c=5)           # TypeError: f() got an unexpected keyword argument 'c'
```

Séquence et paramètres effectifs

```
s = [1, 2, 3]
print(s)   # [1, 2, 3]   1 seul paramètre effectif, la liste s
print(*s)  # 1 2 3       3 paramètres formels (les éléments de s)
```

Xavier Crégut <prenom.nom@enseeiht.fr>

Paramètres seulement nommés (keywork-only argument)

Principe

C'est un paramètre formel qui ne peut pas être initialisé avec un paramètre effectif positionnel mais seulement un paramètre effectif nommé.
Exemple : 'end' et 'sep' de `print`

Syntaxe

```
def f(a, *p, b=3, c):
    print("a = {}, p = {}, b = {}, c = {}".format(a, p, b, c))

f(1, 2, 3, 4)   # TypeError: f() missing 1 required keyword-only argument: 'c'
f(1, 2, 3, 4, c=5)  # a = 1, p = (2, 3, 4), b = 3, c = 5
```

Remarque : Mettre `*` (sans nom) pour ne pas autoriser de paramètres positionnels supplémentaires.

```
def f(a, *, b = 5):
    print(a, b)
```

Effet des appels : `f(1); f(a=2); f(1, 2); f(1, 2, 3); f(a=1, b=2); f(b=2)`

Paramètres seulement positionnels

Principe

C'est un paramètre formel que l'on ne peut initialisé que par un paramètre positionnel et non par un paramètre nommé.

Moyen

Ce sont tous les paramètres qui sont avant un pseudo-paramètre noté « / » (depuis python 3.8).

Exemple

```
def f(a, b=7, /, c=9):
    print(a, b, c)
f(1, 2)       # 1 2 9
f(1)          # 1 7 9
f(1, 2, 3)    # 1 2 3
f(1, 2, c=3)  # 1 2 3
f(1, b=2)  # TypeError: f() got some positional-only arguments passed as keyword arguments: 'b'
```

Intérêt

- Obliger à utiliser les paramètres positionnels
- Le nom du paramètre formel peut être changé sans impact sur les programmes qui l'utilisent

Nombre variable de paramètres nommés

Principe

Le dernier paramètre formel peut être précédé de **.
Il récupère tous les paramètres nommés en surnombre (non associés à un paramètre formel)

Exemple

```
def f(a, /, b=2, c=3, *p, **k):
    print('a={}, b={}, c={}, p={}, k={}'.format(a, b, c, p, k))

f(1)                                # a=1, b=2, c=3, p=(), k={}
f(1, 4)                             # a=1, b=4, c=3, p=(), k={}
f(1, c=5)                           # a=1, b=2, c=5, p=(), k={}
f(9, 8, 7, 6, 5, d=4, e=3)          # a=9, b=8, c=7, p=(6, 5), k={'d': 4, 'e': 3}
f(5, z=1, y=2, a=7)                 # a=5, b=2, c=3, p=(), k={'y': 2, 'z': 1, 'a': 7}
```

Dans le dernier appel, a=7 est considéré comme paramètre nommé en surnombre car le paramètre formel « a » est seulement positionnel et ne peut pas être initialisé de cette manière.
Le nom du paramètre formel « a » pourrait être changé en « aa » et l'appel devrait (et aura) toujours le même résultat.

Excercice

Exercice : Appels de fonctions

Indiquer si les appels suivants sont valides ou non. Si invalides, expliquer pourquoi.

```python
def g(a, /, b, c=4):
    pass

g(1, 2)
g(1, 2, 3)
g(b=2, a=1)
g(b=2, 1)
g(c=2, b=2, a=1)
```

```python
def h(a=0, *b, c):
    pass

h(1, 2, 3, 4)
h(1, 2, c=3)
h(c=3)
h(b=(1, 2), c=4)
```

```python
def k(a, /, b, c=4, **d):
    pass

k(1, 2)
k(1, 2, 3)
k(b=2, a=1)
k(1, b=2, x=5, y=2)
k(1, a=1, z=2, b=2)
k(1, 2, d=7)
```

Xavier Crégut <prenom.nom@enseeiht.fr>

Exercices

1. On considère la fonction `index` qui retourne l'indice de la première occurrence d'un élément `x` dans une séquence `s` à partir de l'indice `début` inclus jusqu'à indice `fin` exclu. Si l'indice fin est omis, on cherche jusqu'au dernier élément de la séquence. Si l'indice de début est omis, la recherche commence au premier élément (indice 0). Donner la signature de cette fonction.
2. Donner la signature d'une fonction max qui retourne le plus grand de plusieurs éléments.
3. Donner la signature de print.
4. Que fait la fonction suivante (on l'expliquera sur l'appel fait) ?

```
def printf(format, /, *args, **argv):
    print(format.format(*args), **argv)

printf("{} -> {}", 'A', 'C', end=' !\n')
```

5. Donner la signature de la fonction range. Dans sa forme générale, elle prend en paramètre l'entier de début, l'entier de fin et un pas. Si le pas est omis, il vaut 1. Si on ne donne qu'un seul paramètre effectif, il correspond à l'entier de fin ; l'entier de début vaut 0 et le pas 1.
6. L'appel `range(stop=5)` provoque `TypeError: range() does not take keyword arguments`.
   Est-ce que ceci remet en cause la signature proposée ?

Mode de passage des paramètres

Questions

1. Quelles modifications peut faire une fonction sur ses paramètres ?
2. L'appelante verra-t-elle ces modifications ?

Exercice : Remplacer les ... pour que le programme s'exécute sans erreur.

comprendre_passage_parametres.py

```python
def f1(s):
    s = ['ok ?']

def f2(s):
    s[0] = '5'
```

```python
liste = [0, 1]
f1(liste)
assert liste == ...

f2(liste)
assert liste == ...
```

```python
couple  = (0, 1)
f1(couple)
assert couple == ...

f2(couple)
assert couple == ...
```

Réponses

Un paramètre formel est un nom associé à l'objet désigné par le paramètre effectif :

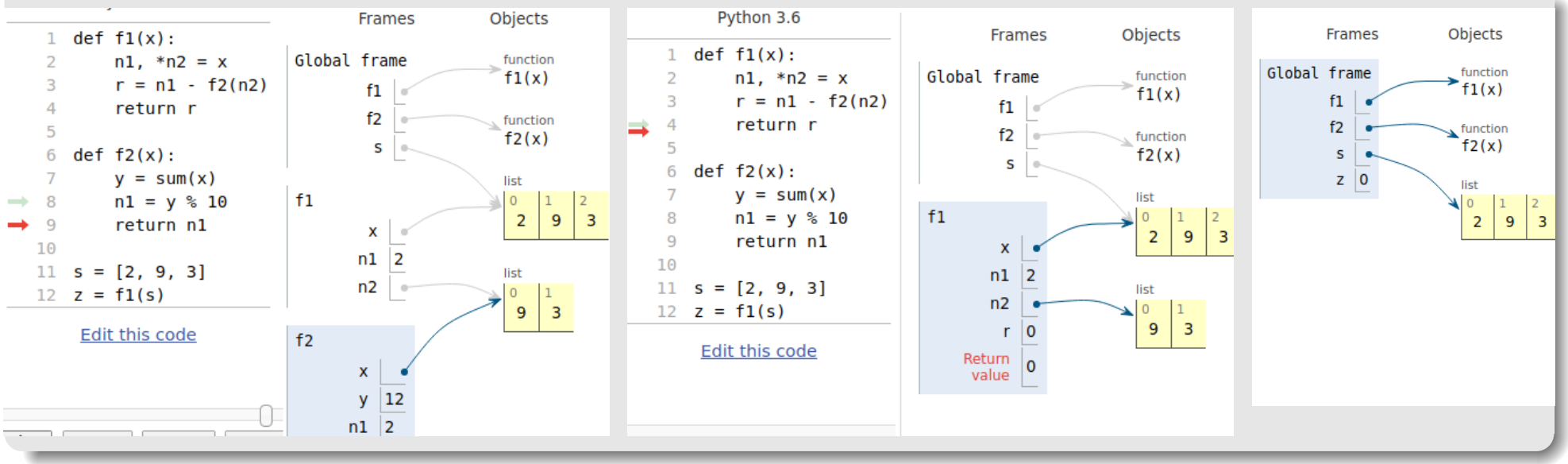
- si l'objet est immuable : aucune modification possible sur l'objet fourni par l'appelant
- si l'objet est muable : toute modification faite par l'appelée sera visible de l'appelant
- changer l'objet associé au paramètre (affecter le paramètre) ne sera pas visible de l'appelant

Exécuter le programme avec Python Tutor pour vérifier les résultats et comprendre ce qu'il se passe.

Variable locale

Principe : Une affectation dans une fonction définit un nouveau nom local (variable) à cette fonction qui disparaîtra quand la fonction se terminera
Une **variable locale** est une variable qui n'existe que pendant l'exécution d'un sous-programme.
Intérêt : Ne pas polluer l'espace des noms avec des données créées dans le corps d'une fonction

comprendre_variable_locale_liste.py



- L'appel de f1(s) créé un contexte pour l'exécution de f1 avec x, n1 et n2 variables locales
- Ligne 3 : l'appel de f2(n2) crée un contexte pour exécution de f2 : x, y et n1 var. locales
- x et n1 de f1 sont indépendants de x et n1 de f2 car les contextes (fonctions) sont différents
- Quand l'exécution de f2 se terminera, x, y et n1 disparaîtront (idem pour f1 et x, n1, n2)
- À la fin ne resteront que f1, f2, s et z (les autres noms ont disparus, pas de pollution)

Espace de noms

Définition

Un **espace de noms** (ou contexte) permet d'associer des objets à des noms.
Les espaces de noms sont hiérarchisés :

- espace de noms **prédéfini** (builtins) : objets prédéfinis.
- espace de noms **global** : noms définis au premier niveau (interpréteur)
- espace de noms d'une fonction : les noms définis dans cette fonction (à l'exécution)

Opérations

- `dir()` affiche les noms de l'espace de noms courant
- `vars()` affiche les noms et les objets associés de l'espace de noms courant
- `globals()` : l'espace de noms global
- `locals()` : l'espace de noms local

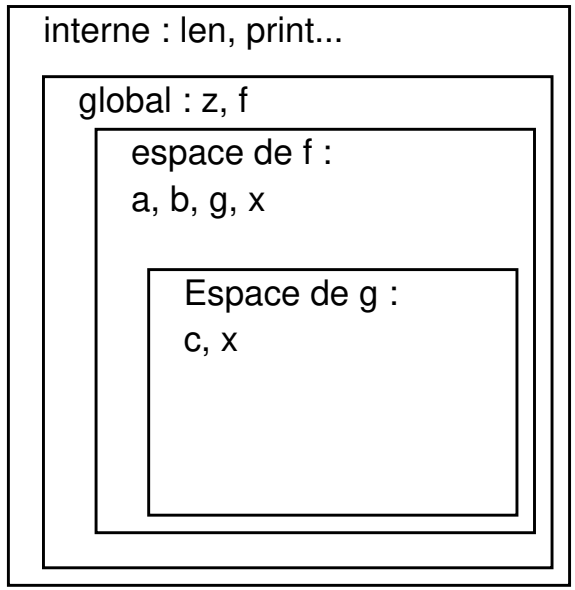
Masquage

Un nom déclaré dans un espace de noms plus interne masque les noms des espaces supérieurs.

Exemple : dans l'exemple qui suit, x défini dans g masque x défini dans f.

Espace de noms : illustration

```
1 z = 0    #! nom 'z' dans l'espace de noms global
2 def f(a, b):     #! nom 'f' dans l'espace de noms global
3                  #! a et b dans l'espace de noms de f
4     def g(c):    #! g dans l'espace de noms de f
5         x = 3
6         print('g/dir  :', dir())
7         print('g/vars :', vars())
8         print('z =', z)  # Quel z ?
9
10    x = 5
11    print('f/dir  :', dir())
12    print('f/vars :', vars())
13    g(a+b)
14
15 print('global/dir  :', dir())
16 print('global/vars :', vars())
17 f(10, 20)
```



montre que x n'existe pas au niveau global, vaut 5 dans f et 3 dans g (masquage du x de f) :

```
global/dir  : [..., 'f', 'z']
global/vars : {..., 'z': 0, 'f': ...}
f/dir  : ['a', 'b', 'g', 'x']
f/vars : {'a': 10, 'b': 20, 'g': ..., 'x': 5}
g/dir  : ['c', 'x']
g/vars : {'c': 30, 'x': 3}
z = 0
```

Hiérarchie des espaces de noms

Exemple

```
def f(a, b):     #! deux paramètres a et b
  def g():
    v = x    #! accède au x de f
    w = z    #! accède au z global
    print('g:', v, id(x), vars())
    pass     #! fin de la portée de v et w
  x = '7'  #! variable locale de f : x
  print('f:', v, id(x), vars())
  g()
  pass  #! fin de la portée de a, b, g et x
v, z = 4, 9   # noms de niveau global
f(1, 2)
```

Principe

- Une fonction définit un espace de noms dans lequel apparaissent :
  - les paramètres
  - les noms locaux (variables, fonctions...)
- Ces noms n'existent que pendant l'exécution de la fonction et disparaissent après.
- Ces noms masquent les noms déclarés dans un espace de noms englobant (v de g).
- Consultation possible des noms des espaces de noms englobants si non masqués (v, x, z).

```
f: 4 140523694628656 {'a': 1, 'b': 2, 'g': ..., 'x': '7'}
g: 7 140523694628656 {'v': '7', 'w': 9, 'x': '7'}
```

Conseil

1. **Une fonction ne doit utiliser que ses paramètres et aucune variable du niveau global**
2. Ne jamais définir des variables au niveau global (risque d'erreurs difficiles à corriger)
   - toutes les variables doivent être déclarées dans des fonctions
   - exception : constante (variable qui ne devra jamais être modifiée), nom en majuscules.

Pour aller plus loin : Nom local et nom global

global

Placé devant un nom, le mot-clé `global` demande à Python d'utiliser le nom global (qui doit donc exister) et donc de ne pas créer une variable locale. **À ne pas utiliser !**

Exemple

```
a = 10       # variable globale
def f():
    a = 5
    print('dans f(), a = ', a)
print(a) ; f() ; print(a)
```

- Que donne cette exécution ?
- Que se passe-t-il si on supprime « `a = 5` » ?
- Que se passe-t-il si on fait « `print(a)` » avant « `a = 5` » ?
- Que se passe-t-il si on ajoute « `global a` » en début de f() ?

Réponse

Exécuter avec python tutor

Récursivité

Définition

Une fonction récursive est une fonction dont l'implantation contient un appel à elle-même.

Important : Il faut donc bien comprendre la spécification de cette fonction !

Exemple : factorielle récursive

```
def fact(n):
    if n <= 1:  #! cas terminal
        return 1
    else:       #! cas général
        return n * fact(n - 1)
```

Exécution

```
fact(4) --> 4 * fact(3) car non 4 <= 1
                \--> 3 * fact(2) car non 3 <= 1
                          \--> 2 * fact(1) car non 2 <= 1
                                    \--> 1 car 1 <= 1
                                    <-- 1
                          <-- 2 * 1 = 2
                \<-- 3 * 2 = 6
       <-- 4 * 6 = 24
```

Xavier Crégut <prenom.nom@enseeiht.fr>

Fondement mathématique

Récurrence

Le corps de la fonction `factorielle` précédente correspond à la définition mathématique de la factorielle donnée sous forme de **récurrence** :

$$n! = \begin{cases} 1 & \text{si } n = 0 \text{ ou } n = 1 \\ n \times (n-1)! & \text{sinon} \end{cases}$$

```
def fact(n):     #! récursive
    if n <= 1:
        return 1
    else:
        return n * fact(n - 1)
```

Autre formulation mathématique

On pourrait définir la factorielle par un produit conduisant à une version non récursive.

$$n! = \prod_{i=1}^{i=n} i = 1 \times 2 \times 3 \ldots \times (n-1) \times n$$

```
def factorielle(n):     #! itérative
    resultat = 1
    for k in range(2, n+1):
        resultat = resultat * k
    return resultat
```

Les définitions par récurrence (et donc les SP récursifs) sont souvent plus concises et claires que leurs équivalents itératifs.

Xavier Crégut <prenom.nom@enseeiht.fr>

Terminaison

Motivation

Il faut être sûr que les appels récursifs s'arrêtent.
En Python : RecursionError: maximum recursion depth exceeded

Terminaison

- Prévoir un (ou plusieurs) cas de base (terminal) sans appel récursif.
- Dans le cas général, mettre en évidence un entier positif (taille du problème) qui décroit strictement à chaque appel récursif.

Application à la factorielle :

- On définit la taille du problème de fact(n) comme étant `n`
- le cas terminal correspond à $n \leq 1$
- dans le cas général ($n > 1$), l'appel récursif est `fact(n - 1)` de taille strictement inférieure ($n - 1 < n$).

Récursivité terminale

Définition

Une fonction est **récursive terminale** quand le résultat de l'appel initial est directement celui du dernier appel récursif.

Règle

Aucune opération n'est réalisée sur le retour d'un appel récursif.

Factorielle en récursivité terminale

```
def fact(n, r):     #! récursivité terminale
    if n <= 1:
        return r
    else:
        return fact(n - 1, n * r)
```

Le calcul de factorielle de 4 :

```
fact(4, 1) --> fact(3, 4) --> fact(2, 12) --> fact(1, 24) --> 24
         <--24--         <--24--          <--24--           <--24--
```

Toute fonction récursive terminale peut être réécrite avec une répétition

```python
#! récursivité terminale
def fact_t(k, r):
    if k <= 1:
        return r
    else:
        return fact_t(k - 1, k * r)

def fact(n):
    return fact_t(n, 1)

f = fact(4)
```

```python
#! version itérative
def fact(n):
    k = n
    resultat = 1
    while k > 1:
        resultat = k * resultat
        k = k - 1
    return resultat


f = fact(4)
```

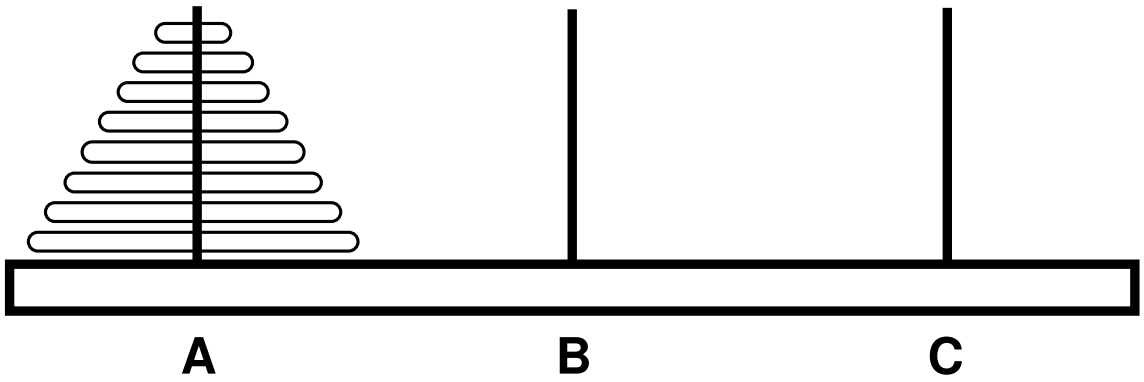
Correspondances

- resultat est équivalent à r
- resultat est initialisé à 1, valeur de r lors du premier appel à fact_t
- la condition du `while` correspond à la condition du cas général cas général (k > 1)
- les instructions du `while` correspondent à l'appel récursif
  - le nouveau resultat est k * resultat
  - le nouveau k est k - 1
- le `while` devrait ici être replacé par un `for`

Exercice : Les tours de Hanoï

C'est Lucas, mathématicien français, qui sous le pseudonyme de Claus a inventé ce jeu.

Il se présente sous la forme d'un support en bois sur lequel sont plantées trois tiges A, B et C. Sur ces trois tiges peuvent être enfilés des disques de diamètres différents (8 dans la version originale, mais N de manière générale). Dans la configuration initiale, les disques sont empilés par ordre de taille décroissante sur la tige A.



Le but est de déplacer tous les disques de la tige A vers la tige C sachant qu'on ne peut déplacer qu'un seul disque à la fois et qu'on n'a pas le droit de le déposer sur un disque plus petit que lui.

Écrire un programme qui donne la solution de ce jeu (c'est-à-dire, la liste des coups à jouer). Pour cela, on remarquera qu'un coup est complètement déterminé par la donnée de la tige de départ et de la tige d'arrivée, car on ne peut déplacer qu'un disque à la fois (celui au sommet de la tige).

Induction

Principe

L'induction est une généralisation de la récurrence qui consiste à s'appuyer sur la structure de l'information pour déterminer les cas à traiter.

Exemples

- Sur les entiers : on raisonne sur l'entier et son suivant (principe de la récurrence).
  - cas de base n = 0
  - cas général : suivant de n (donc n + 1).
- Sur une liste
  - cas d'une liste vide
  - cas général : un premier élément et les autres éléments
- Sur un arbre binaire (deux fils au plus), une liste de listes :
  - cas d'un arbre vide (liste vide)
  - cas général : un noeud avec sa valeur, ses sous-arbres gauche et droit
- Sur les chiffres d'un nombre
  - on s'intéresse aux chiffres d'un nombre
  - quelle induction ?

Application :

- Somme des entiers de 0 à n (10 -> 55).
- Somme des nombres d'une liste ([1, 4, 0, 2] -> 7), des nombres d'un arbre ([1, [2, 0, [5]], [6]] -> 14), des chiffres d'un entier naturel (3505 -> 13).

Intérêt des sous-programmes

1. **Structuration de l'algorithme :**
   - les sous-programmes correspondent généralement aux actions complexes du raffinage
   - ces actions complexes apparaissent donc clairement
2. **Compréhensibilité :**
   - découpage d'un algorithme en « morceaux »
   - lecture à deux niveaux : 1) la spécification et 2) l'implantation
   - la spécification est suffisante pour comprendre l'objectif d'un sous-programme (et savoir l'utiliser)
3. **Factorisation et réutilisation**
   - un sous-programme évite de dupliquer du code
   - il peut être réutilisé dans ce programme et dans d'autres (modules)
4. **Mise au point facilitée**
   - tester individuellement chaque sous-programme avant le programme complet
   - erreurs détectées plus tôt, plus près de leur origine, plus faciles à localiser et corriger
5. **Amélioration de la maintenance :**
   - car le programme est plus facile à comprendre
   - l'évolution devrait rester localisée à un petit nombre de sous-programmes

Étapes pour définir un sous-programme

1. **Définir la spécification du sous-programme**
   a. Définir l'objectif du SP (équivalent R0)
   b. Donner des exemples d'utilisation du SP
   c. Identifier les paramètres du sous-programme : rôle puis nom, puis le mode et le type
   d. Choisir un nom significatif pour le sous-programme
      - verbe à l'infinitif si procédure (retourne toujours `None`)
      - mot significatif qui décrit le résultat (candidat : le nom du paramètre en `out`)
   a. Lister les conditions d'applications (formalisées par les préconditions)
   b. Expliquer l'effet du sous-programme (formalisés par les postconditions)
   c. Rédiger la spécification du sous-programme à partir de ces informations (signature + docstring)
2. **Écrire des programmes de test (test unitaire)**
3. **Définir l'implantation du sous-programme**
   - Appliquer la méthode des raffinages : la spécification du SP est le R0
   - Chaque action complexe identifiée est candidate à être un SP
4. **Tester l'implantation du sous-programme**
   - corriger le sous-programme (ou les tests) en cas d'échec et rejouer tous les tests

Conseils sur la définition de sous-programmes

- Dans un même SP, **ne pas mélanger traitement** (calcul) **et interactions avec l'utilisateur** (affichage ou saisie) :
  - Les traitements sont plus stables que l'IHM (interface humain-machine).
  - L'interface utilisateur peut changer, il peut y en avoir plusieurs (texte, graphique, web, smartphone...)
- Un SP doit **être une boîte noire** :
  - il doit être indépendant de son contexte
  - ...donc on devrait pouvoir le copier/coller[6] dans un autre contexte
  - ...il ne doit pas pas dépendre de variables globales
- Un SP ne doit **pas avoir trop de paramètres**
  - soit mauvais découpage,
  - soit regrouper les paramètres avec un nouveau type (liste, tuple, classe...)
- Un sous-programme **doit être court**. Il faut le découper en sous-programmes si :
  - il est trop long ($>$ 20 lignes, arbitraire !)
  - il est trop complexe (trop de structures de contrôle imbriquées)
- On doit **être capable d'exprimer clairement l'objectif du SP** (docstring)...
  ...sinon c'est qu'il est mal compris !

6. C'est une image ! Il ne faut jamais faire de copier/coller !

Sommaire

Modules : réutilisation entre programmes

Définition

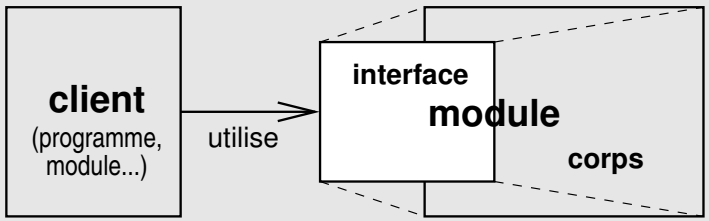
Regroupement de plusieurs définitions (noms, fonctions, classes, etc.) sur un même thème.

Exemples

1. Un module *robots* définissant le type *Robot* et les sous-programmes associés.
   - Avoir un type *Robot* permet de gérer simultanément plusieurs robots.
2. Un module *math* qui regroupe les fonctions mathématiques
3. ...

Intérêts

- **structurer** une application en sous-parties cohérentes
- **réutiliser** les définitions du module dans différents programmes
- **définir un espace de nom** (deux modules différents peuvent utiliser les mêmes noms)
- **cacher** les détails de réalisation
  - **interface** (ou spécification) du module connue (publique)
  - **corps** (ou implantation) du module cachée
  - Tout ce qui est caché peut être changé sans impact sur les utilisateurs du module (favorise l'évolutivité)



Xavier Crégut <prenom.nom@enseeiht.fr>

Utiliser des définitions d'autres modules

Principe

Pour utiliser une définition d'un autre module, il faut le dire explicitement à Python (`import`)

- importer le module en entier. On utilise la notation pointée pour accéder à son contenu
- importer une ou plusieurs définition d'un module
- possibilité de changer localement le nom de l'élément importé

Importer un module

```
import math, random as alea       # On donne simplement accès aux noms `math` et `ran

print(math.cos(math.pi / 3))     # définition préfixée du nom du module
print(alea.randint(1, 100))
```

Importer les définitions d'un module

```
from math import pi, cos, pow          # accès à pi, cos et pow de math
from random import randint as alea     # accès à randint sous le nom alea

print(cos(pi / 3))       # pi directement accessible
print(alea(1, 100))
# math.pi ~~>  NameError: name 'math' is not defined
```

Xavier Crégut <prenom.nom@enseeiht.fr>

Définir un module en Python

Définition

Un **module** Python est un fichier (extension .py) :

- le nom du fichier (sans .py) est le nom du module
- il contient des instructions :
  - définitions de noms : variables, fonctions, ...
  - instructions d'initialisation du module
- ces instructions sont exécutées une seule fois lors du chargement du module

Convention

Un nom préfixé par un « _ » signifie qu'il est local, privé au module (niveau implantation) :

- il ne devrait pas être utilisé à l'extérieur du module
- car il est susceptible d'être modifié, supprimé, etc.
- il ne sera pas accessible quand on fait un `from m import *`

Mais rien n'empêche de l'utiliser.

Principe : « Nous sommes entre personnes responsables »

Exemple

- Le module random définit `_cos` et `_pi` : une fonction et une variables privées
- On faisant `from random import *`, on n'y a pas accès (_cos : NameError)
- On peut faire `from random import _pi, _cos`
- On peut aussi faire `import random`, puis `random._cos`

Module ou programme principal

Principe

Un même fichier Python peut être utilisé à la fois comme :

- module (et donc importé par d'autres modules et programmes)
- programme

Principe

Le nom prédéfini `__name__`, initialisé par l'interpréteur Python, dit comment a été chargé le fichier :

- s'il est exécuté comme programme, `__name__` est `"__main__"`
- s'il est importé, `__name__` est le nom du module (donc du fichier)

Exemple

- Le nom `__name__` est intialisé avec `__main__` si le fichier est exécuté comme un programme

m1.py

```
print('__name__ dans m1 :', __name__)
```

prog.py

```
import m1
print('__name__ dans prog :', __name__)
```

```
$ python m1.py
__name__ dans m1 : __main__
$ python prog.py
__name__ dans m1 : m1
__name__ dans prog : __main__
```

Conseil : Mettre les instructions d'un programme dans un `if __name__ == '__main__':`

Exemple : le module arithmetique ne contenant que la fonction pgcd

arithmetique.py

```python
'''Exemple de module avec un seul sous-programme. '''

def pgcd(a: int, b: int) -> int:
    '''Le pgcd, plus grand commun diviseur, de a et b, entiers naturels.

    Les entiers a et b doivent être strictement positifs.
    Cette version est naïve et donc peu efficace.

    :param a, b: deux entiers strictement positifs.
    :pre: a > 0 and b > 0
    :returns: le pgcd de a et b.
    :post: a % result == 0 and b % result == 0 and "c'est le plus grand"

    >>> pgcd(21, 15)
    3
    '''
    assert a > 0, f'a doit être > 0 : a == {a}'
    assert b > 0, f'b doit être > 0 : b == {b}'

    while a != b:
        # soustraire au plus grand le plus petit
        if a > b:
            a = a - b
        else:
            b = b - a
    return a
```

Utilisation de la fonction pgcd

exemple_pgcd.py

```
1  '''Programme utilisant le pgcd'''
2
3  from arithmetique import pgcd
4
5  def main():
6    '''Afficher le pgcd de 2 entiers
7    lus au clavier. Non robuste.'''
8    # demander deux entiers a et b
9    a = int(input('a = '))
10   b = int(input('b = '))
11
12   # calculer le pgcd de a et b
13   p = pgcd(a, b)
14
15   # afficher le pgcd
16   print('pgcd =', p)
17
18 if __name__ == '__main__':
19     main()
```

Cas nominal

```
$ python exemple_pgcd.py
a = 15
b = 20
pgcd = 5
```

Cas d'erreur :

```
$ python exemple_pgcd.py
a = 15
b = 0
Traceback (most recent call last):
  File "exemple_pgcd.py", line 19, in <module>
    main()
  File "exemple_pgcd.py", line 13, in main
    p = pgcd(a, b)
  File "arithmetique.py", line 18, in pgcd
    assert b > 0, f'b doit être > 0 : b == {b}'
AssertionError: b doit être > 0 : b == 0
```

Explications

- utilisation de **import** car pgcd est défini dans un autre fichier (module)
- cas nominal : « **return** a » termine la fonction et la valeur de a est associée à p (ligne 13)
- cas d'erreur : **assert** termine la fonction et provoque l'arrêt du programme (voir exceptions)

Sommaire

doctest : Exécuter les exemples de la spécification

```python
def statistiques(donnees):
    """Calculer le min et le max de donnees.

    :param donnees: les données à traiter
    :returns: (min, max) : quelques statistiques sur donnees ou None si pas de données

    >>> statistiques([1, 3, 2])   #! un premier exemple
    (1, 3)                        #! le résultat attendu

    >>> statistiques([])          #! un deuxième exemple
    (None, None)                  #! le résultat attendu
    """
    vmin = vmax = None # le min et max des valeurs traitées
    for x in donnees:
        # mettre à jour les statistiques
        if vmin is None:
            vmin = vmax = x
        elif x > vmax:
            vmax = x
        elif x < vmin:
            vmin = x
    return (vmin, vmax)


if __name__ == "__main__":
    # Vérifier les exemples
    import doctest
    doctest.testmod()
```

- donner des exemples complète la spécification
  - donner des exemples est un moyen de spécifier
- **doctest** exécute les exemples de la *docstring*
  - instructions après >>> fournies à l'interpréteur
  - le résultat attendu est sur la ligne suivante
- compare le résultat de l'interpréteur au résulat attendu
  - fragile car comparaison de chaînes : espaces, etc.
- signale les exemples qui sont faux
- n'affiche rien si tout est ok : pas rassurant !
- permet de **valider les exemples fournis**
- mais **insuffisant pour faire du test**

Tester avec pytest

Constat : On ne peut pas mettre tous les tests dans la spécification des sous-programmes !

Principe de pytest :

- écrire les tests dans des fichiers `*_test.py` ou `test_*.py`
- les fonctions qui commencent par 'test' sont considérées comme des programmes de test
- **assert** est utilisé pour savoir si le test réussit ou échoue

test_statistiques.py

```
1 from statistiques import statistiques
2
3 def test_statistiques_nominaux():
4     assert statistiques([1, 3, 2]) == (1, 3)
5
6 def test_statistiques_nominaux2():
7     assert statistiques([1, 3, 2, 4, 1]) == (1, 4)
8
9 def test_statistiques_limite():
10    assert statistiques([]) == (None, None)
11
```

Lancer les tests : pytest

- lance tous les tests trouvés
- l'option `--doctest-modules` permet de tester aussi les exemples des docstrings

Conclusion : Préférer pytest (sait exécuter les tests unittest) !

Compléments pytest

- `@pytest.fixture` : définir un contexte de test
  - c'est une fonction ; son résultat est une donnée de test
  - tout paramètre dont le nom est celui d'un contexte de test est initialisé avec le résultat de ce contexte
  - **intérêt :** factorise l'initialisation de la donnée
- `pytest.raises(ExpectedException)` : vérifie que le test lève bien l'exception attendue

test_list.py

```python
import pytest

@pytest.fixture
    # contexte de test :
    # un paramètre appelé liste1
    # sera initialisé avec liste1()
def liste1():
    return [1, 5, 3, -8, 12]

def test_len(liste1):
        # liste1 sera initialisé
        # avec le résultat de liste1()
    assert 5 == len(liste1)

def test_contains(liste1):
    assert 1 in liste1
    assert -8 not in liste1
    assert 10 in liste1

def test_append(liste1):
    liste1.append(7)
    assert [1, 5, 3, -8, 12, 7] == liste1
    l = []
    l.append(1)
    assert [1] == l

def test_index(liste1):
    assert 0 == liste1.index(1)
    assert 3 == liste1.index(-8)
    assert 4 == liste1.index(12)

def test_index_non_trouve(liste1):
    with pytest.raises(ValueError):
        liste1.index(2)

def test_avec_erreur(liste1):
    liste1.index(2)
```

Le résultat de l'exécution :

```
> pytest test_list.py test_statistiques.py
============================= test session starts ==============================
platform linux -- Python 3.10.12, pytest-7.4.0, pluggy-1.3.0
rootdir: /home/cregut/projects/ens/python/cours/exemples
plugins: timeout-2.3.1
collected 9 items
test_list.py .F...F                                                      [ 66%]
test_statistiques.py ...                                                 [100%]
================================== FAILURES ===================================
_____________________________ test_contains ______________________________
liste1 = [1, 5, 3, -8, 12]
    def test_contains(liste1):
        assert 1 in liste1
>       assert -8 not in liste1
E       assert -8 not in [1, 5, 3, -8, 12]
test_list.py:17: AssertionError
_____________________________ test_avec_erreur ____________________________
liste1 = [1, 5, 3, -8, 12]
    def test_avec_erreur(liste1):
>       liste1.index(2)
E       ValueError: 2 is not in list
test_list.py:37: ValueError
=========================== short test summary info ============================
FAILED test_list.py::test_contains - assert -8 not in [1, 5, 3, -8, 12]
FAILED test_list.py::test_avec_erreur - ValueError: 2 is not in list
========================= 2 failed, 7 passed in 0.03s ==========================
```

Remarque : L'instruction `assert 10 in liste1` n'a pas été évaluée car la précédente a échoué.

Couverture des tests : coverage.py

- Essentiel en Python : l'interpréteur ne vérifie que la syntaxe.
  $\Longrightarrow$ Il faut exécuter le code pour trouver les autres erreurs !
- Conséquence : S'assurer qu'il y a une couverture de 100 % de chaque instruction
  - C'est une condition nécessaire, pas suffisante !
- coverage.py fait des calculs de couverture : instructions et enchaînements
- Installation : `pip install coverage`
- Faire un calcul de couverture des tests unitaires : `coverage run -m py.test`
- ou pour un programme particulier : `coverage run exemple_pgcd.py`
- L'option `--branch` ajoute le calcul des enchaînements : `coverage run --branch -m py.test`
- Produire un rapport (`-m`, ou `--show-missing`, pour indiquer les lignes non exécutées) :

```
$ coverage report -m statistiques.py
Name                Stmts   Miss Branch BrPart  Cover   Missing
-----------------------------------------------------------------
statistiques.py        13      3     10      2    78%   21, 27-28
```

- ou en HTML : `coverage html`

Sommaire

Exceptions : on les rencontre vite !

Motivation

Que peut faire :

- l'interpréteur Python si on lui demande d'interpréter un programme avec une erreur de syntaxe ou une variable non définie ?
- la méthode `index` de string si on fournit une chaîne qui n'existe pas ?

Le travail ne peut pas être fait. Une exception le signale : SyntaxError, NameError, ValueError...

Exemples

```
>>> 'bonjour'.index('j')
3
>>> 'bonjour'.index('z')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

```
>>> xxx
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'xxx' is not defined
```

Définition

Exception = moyen pour une fonction de signaler que le travail attendu ne peut pas être réalisé.

Lever une exception

Syntaxe sur un exemple

```
>>> raise ValueError("Ma première exception")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Ma première exception
```

Explications

- `raise` est le mot-clé qui permet de lever une exception
- `ValueError` est l'exception levée
- entre parenthèses, on peut indiquer un message (transmis avec l'exception)

Remarque

La levée d'une exception se fait normalement dans un sous-programme.
C'est le moyen de signaler les éventuelles anomalies à l'appelant.

Définir sa propre exception (MonException)

```
class MonException(Exception):
    pass
```

Traiter une exception : un exemple

Exception vs Conditionnelle

```
s = 'bonjour'
c = 'z' # ou 'j', 'r'...
# indice de c dans s
try:
    print(s.index(c))
except ValueError:
    print("aucun")
```

```
s = 'bonjour'
c = 'z' # ou 'j', 'r'...
# indice de c dans s
if c in s:
    print(s.index(c))
else:
    print("aucun")
```

Discussion

Avec les exceptions : approche optimiste

- on écrit une version optimiste du code (où tout se passe bien) : dans un `try`
- on traite ensuite les cas d'erreurs : dans les `except` associés
- le code nominal est plus facile à lire
- si on sait traiter l'exception, on la récupère sinon elle continue à se propager
- Risque : mal identifier l'origine de l'exception et faire le mauvais traitement

Avec une conditionnelle : approche pessimiste

- tester explicitement tous les cas anormaux
- le code peut devenir difficile à lire
- peut être plus coûteux (ici, deux parcours de s : un pour `in`, l'autre pour index)
- Remarque : le test pourrait être fait *a posteriori* sur le résultat de la fonction

Exemples

La fonction isqrt

isqrt.py

```python
def isqrt(n):
    '''
    Retourner la racine carrée entière de n.
    Exceptions : ValueError si n est strictement négatif.
    '''
    if n < 0:
        raise ValueError(f"Non positif: {n}")
    return int(n ** .5)
```

La fonction index

index.py

```python
def index(sequence, element):
    '''
    Retourne le plus petit indice de element dans sequence.
    Exception IndexError si element n'est pas dans sequence.
    '''
    for i, x in enumerate(sequence):
        if x == element:
            return i
    else:
        raise IndexError(f"{element} not found")
```

- Lever l'exception en début de fonction quand c'est possible (pas de `else`).

Utilisation de isqrt

main_isqrt.py

```python
import sys                  #! documentation omise
from isqrt import isqrt

def somme_isqrt(sequence):
    return sum(isqrt(x) for x in sequence)

def main():
    try:
        somme = somme_isqrt(float(x) for x in sys.argv[1:])
        print("Somme isqrt:", somme)
    except ValueError as e:
        print("Un argument n'est pas positif.")
        print(e)

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

```
> python main_isqrt.py 4 1 11
Somme isqrt: 6
> python main_isqrt.py 4 -1 1 -2
Un argument n'est pas positif.
Non positif: -1.0
```

1. Qu'elle devrait être la documentation de `somme_isqrt` ?
   - retourne la somme des racines carrées entières des nombres de la séquence
   - exception ValueError si l'un des nombres n'est pas positif
2. Qu'elle devrait être la documentation de `main` ?
   - retourne la somme de racines carrées entières des paramètres de la ligne de commandes.
   - Afficher un message si l'un des arguments n'est pas positif.
3. En général, une fonction lève une exception, une autre la récupère.

Mécanisme d'exception

Principe

Mécanisme en trois phases :

1. **Levée** de l'exception quand une anomalie est détectée : `raise`
   L'exécution du bloc est interrompue et l'exception commence à se *propager*.
2. **Propagation** de l'exception : (automatique) l'exception remonte blocs et sous-programmes
3. **Récupération** de l'exception : `except`
   fait par la portion de code qui sait traiter
   reprise de l'exécution normale
   toujours associée à un `try` : seules les exceptions levées dans ce `try` peuvent être récupérées

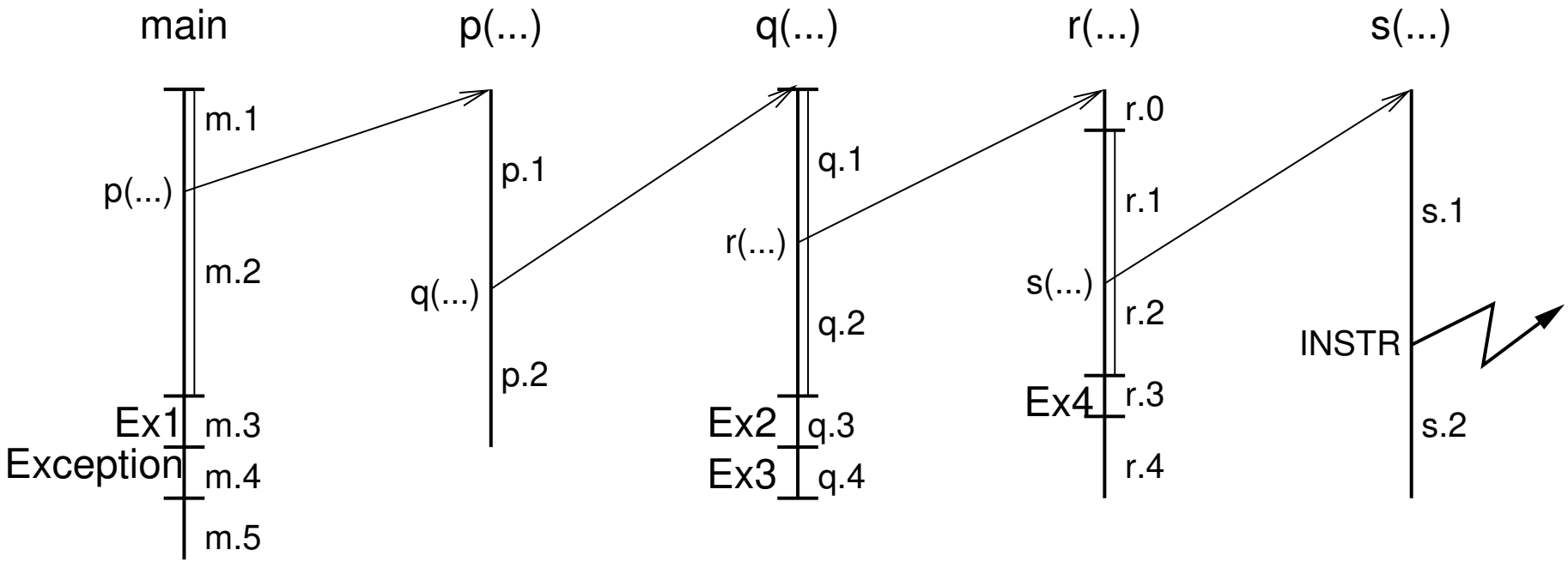
Intérêt

- découpler la partie du programme qui détecte une anomalie de la partie qui sait la traiter
  - plus pratique qu'un code d'erreur !
- permettre d'écrire un code optimiste (plus lisible) en regroupant après le traitement des erreurs

`raise` vs `return` dans une fonction

- `raise` et `return` arrêtent l'exécution de la fonction
- `return` rend la main à l'appelant de cette fonction
- `raise` « saute » tous les appelants jusqu'au prochain `except` correspondant à l'exception

Mécanisme de propagation d'une exception



Exercice : Indiquer la suite de l'exécution de ce programme lorsque instr lève Ex4, Ex3, Ex1, Ex5 ou Err (dire les blocs de code exécutés, les blocs sont nommés lettre.chiffre : m.1, m.2. . .).

Le programme correspondant en Python

```python
def main(nom):
    try:
        print("m.1", end=' ')
        p(nom)
        print("m.2", end=' ')
    except Ex1:
        print("m.3", end=' ')
    except Exception:
        print("m.4", end=' ')
    print("m.5")

def p(nom):
    print("p.1", end=' ')
    q(nom)
    print("p.2", end=' ')

def q(nom):
    try:
        print("q.1", end=' ')
        r(nom)
        print("q.2", end=' ')
    except Ex2:
        print("q.3", end=' ')
    except Ex3:
        print("q.4", end=' ')

def r(nom):
    print("r.0", end=' ')
    try:
        print("r.1", end=' ')
        s(nom)
        print("r.2", end=' ')
    except Ex4:
        print("r.3", end=' ')
    print("r.4", end=' ')

def s(nom):
    print("s.1", end=' ')
    INSTR(nom)
    print("s.2", end=' ')


def INSTR(nom):
    print("INSTR", end=' ')
    if nom == "Ex1": raise Ex1()
    if nom == "Ex2": raise Ex2()
    if nom == "Ex3": raise Ex3()
    if nom == "Ex4": raise Ex4()
    if nom == "Ex5": raise Ex5()
    if nom == "Err": raise Err()

```

Résultat de l'exécution

```
main('---') ->  m.1 p.1 q.1 r.0 r.1 s.1 INSTR s.2 r.2 r.4 q.2 p.2 m.2 m.5
main('Ex4') ->  m.1 p.1 q.1 r.0 r.1 s.1 INSTR r.3 r.4 q.2 p.2 m.2 m.5
main('Ex3') ->  m.1 p.1 q.1 r.0 r.1 s.1 INSTR q.4 p.2 m.2 m.5
main('Ex1') ->  m.1 p.1 q.1 r.0 r.1 s.1 INSTR m.3 m.5
main('Ex5') ->  m.1 p.1 q.1 r.0 r.1 s.1 INSTR m.4 m.5
main('Err') ->  m.1 p.1 q.1 r.0 r.1 s.1 INSTR Traceback (most recent call last):
  File "exception_comprendre.py", line 72, in <module>
    lanceur()
  File "exception_comprendre.py", line 70, in lanceur
    print("main('Err') -> ", end=' '); main("Err")
  File "exception_comprendre.py", line 4, in main
    p(nom)
  File "exception_comprendre.py", line 14, in p
    q(nom)
  File "exception_comprendre.py", line 20, in q
    r(nom)
  File "exception_comprendre.py", line 31, in r
    s(nom)
  File "exception_comprendre.py", line 39, in s
    INSTR(nom)
  File "exception_comprendre.py", line 50, in INSTR
    if nom == "Err": raise Err()
__main__.Err
```

Structure générale de la récupération d'une exception

Syntaxe

```
try:
    # lignes qui peuvent lever une exception
    # normalement au travers de l'appel de sous-programmes
except TypeException1:
    # traitement de l'exception TypeException1
    # On n'utilise pas l'objet exception récupéré
except TypeException2 as nom_exception:
    # traitement de l'exception TypeException2
    # nom_exception référence l'objet exception qui se propageait
    ...
except Exception as e: # on récupère (presque) toutes les exceptions !
    # traitement...
else:
    # Ne sera exécuté que si aucune exception n'est levée
finally:
    # Sera toujours exécuté, qu'il y ait une exception ou pas,
    # qu'elle soit récupérée ou pas
```

Explications

Dans les commentaires ci-dessus ;-)

Quelques exceptions prédéfinies de Python

Le code

```
try:
    indice = chaine.index(sous_chaine)
except NameError as e:
    print('un nom est inconnu :', e)
except AttributeError as e:
    print("Certainement 'index' qui n'est pas trouvé :", e)
except TypeError as e:
    print("Certainement un problème sur le type de 'chaine' :", e)
except ValueError as e:
    print("Certainement la sous-chaîne qui n'est pas trouvée :", e)
except Exception as e:
    print("C'est quoi celle là ?", e)
else:
    print("Super, pas d'exception levée.  L'indice est", indice)
finally:
    print("Je m'exécute toujours !")
```

Les consignes

- Mettre ce code dans un fichier exemple_exception.py (par exemple) et l'exécuter.
- Ajouter en début `chaine = 10` et exécuter
- Changer `10` en `'bonjour'` et exécuter
- Ajouter `sous_chaine = 0` et exécuter
- Changer `0` en `'z'` et exécuter
- Changer `'z'` en `'j'` et exécuter

`with ... [as ...]`

Lire et afficher le contenu d'un fichier texte

```
nom_fichier = 'exemple.txt'
with open(nom_fichier, 'r') as fichier:
    for ligne in fichier:   # pour chaque ligne de fichier
        print(ligne, end='')    # le '\n' est déjà dans `ligne`
```

Explications

- `with` garantit que l'objet créé (ici le fichier) sera bien fermé $\Longrightarrow$ **Utiliser** `with`
- `fichier.readline()` : retourne une nouvelle ligne du fichier (`''` quand plus de ligne à lire)
- `fichier.readlines()` : retourne la liste de toutes les lignes du fichier

`with ...` est transformé en `try ... finally`

```
try:
    fichier = None
    fichier = open(nom_fichier, "r")
    for ligne in fichier:
        print(ligne, end='')
finally:
    if fichier is not None:
        fichier.close()
```

Exemple : Écrire dans un fichier texte

Écrire dans un fichier

```
nom_fichier = '/tmp/exemple.txt'
with open(nom_fichier, 'w') as fichier:
    fichier.write('Première ligne\n')   # `write` n'ajoute pas de '\n'
    print('Deuxième ligne', file=fichier)
```

Remarque

- `write()` retourne le nombre de caractères effectivement écrits.

Sommaire

Motivation

Un programme manipule des données et généralement des **groupes de données**.

Il est donc important de disposer de groupes de données équipés des **opérations et** du **comportement** appropriés pour une classe de problèmes :

- connaître le nombre de caractères différents utilisés dans un texte
- connaître la fréquence (le nombre d'occurrences) des mots d'un texte
- enregistrer les demandes d'impressions reçues par une imprimante
- ...

Et, bien sûr, ces opérations doivent être **efficaces** !

Rappels

On a déjà vu :

- séquence immuable :
  - n-uplet (**tuple**)
  - chaîne (**str**)
- séquence modifiable :
  - liste (**list**)

Voir Data Structures in The Python Tutorial

Voir collections.abc – Abstract Base Classes for Containers : hiérarchie des types et opérations disponibles.

Séquence : opérations prédéfinies

Séquence : Éléments contigus repérés par une position, un indice entier (0 pour le premier élément)

Opération	Résultat	Exemples
len(s)	la longueur de s (nombre d'éléments)	`len('bonjour') == 7`
s[i]	ième élément de s, i == 0 pour le premier	`(1, 5,3)[0] == 1`
	IndexError si `not (-len(s) <= i < len(s))`	`(1, 5,3)[3]` ⇝ IndexError
x in s	True si x est un élément de s	`5 in (1, 5, 3)`
x not in s	True si x n'est pas un élément de s	`4 not in (1, 5, 3)`
s + t	concaténation de s avec t	`(1, 2) + (3,) == (1, 2, 3)`
s * n ou n * s	n concaténations de s avec elle-même	`'x' * 3 == 3 * 'x'== 'xxx'`
min(s)	plus petit caractère de s	`min('bonjour') == 'b'`
max(s)	plus grand caractère de s	`max('bonjour') == 'u'`
s.count(x)	nombre total d'occurrence de x dans s	`'bonjour'.count('o') == 2`
s.index(x[, d[, f]])	indice de la première occurrence de x dans s	`'bonjour'.index('o') == 1`
	(à partir de l'indice d et avant l'indice f)	`'bonjour'.index('o', 2) == 4`
	lève l'exception ValueError si x non trouvé	`'bonjour'.index('x')` ⇝ ValueError
	opérations sur séquences muables	avec `s = [1, 2, 1]`
s.append(x)	ajoute x à la fin de s	`s.append(3); s == [1, 2, 1, 3]`
s.clear()	supprime tous les éléments de s	`s.clear(); s == []`
s.copy()	copie superficielle de s	`t = s.copy(); t == s and t is not s`
s.extent(t)	concatène t à la fin de s (s += t)	`s.extent([3, 5]); s == [1, 2, 1, 3, 5]`
s.insert(i, x)	insère x dans s à l'indice i	`s.insert(1, 3); s == [1, 3, 2, 1]`
x = s.pop(i)	supprime et retourne l'élément à l'indice i	`x = s.pop(1); s == [1, 1] and x == 2`
s.remove(x)	supprime le premier x de s	`s.remove(1); s == [2, 1]`

- **Remarque :** Toutes ces opérations sont présentes sur toute séquence.

Xavier Crégut <prenom.nom@enseeiht.fr>

Pile (Stack)

Les opérations classiques sont :

- empiler : ajouter un nouvel élément en sommet de pile
- dépiler : supprimer l'élément en sommet de pile
- sommet : consulter l'élément en sommet de pile
- vide : savoir si la pile est vide

C'est une structure de données de type **LIFO** : Last In, First Out.

Le type **Stack n'existe pas** en Python mais **list** permet de le réaliser simplement.

- les éléments sont ajoutés en sommet de pile : **list.append()**
- on récupère et supprime l'élément au sommet de la pile : **list.pop()**

```
pile = [1, 2, 3]      # [1, 2, 3], 3 au sommet
pile.append(4)        # [1, 2, 3, 4] : 4 est le nouveau sommet
pile.append(5)        # [1, 2, 3, 4, 5] : 5 est le nouveau sommet
pile[-1]              # obtenir le sommet, la pile n'est pas modifiée
x = pile.pop()        # supprime l'élément en sommet et le retourne : 5
pile                  # [1, 2, 3, 4]
```

File (Queue)

Caractéristiques d'une file (**FIFO** : First In First Out), similaire à une file d'attente classique :

- les éléments sont **ajoutés** à la fin de la file
- les élément sont **extraits** par le début de la file

On pourrait par exemple réaliser une file avec une liste en utilisant :

- file.append(x) : pour ajouter à la fin de la file
- x = file[0] : pour consulter l'élément au début de la file
- x = file.pop(0) : pour extraire l'élément au début de la file (et le retourner)

Mais ce serait **peu efficace** (pour la suppression d'un élément)

Il existe le type **deque** (Double Ended QUEue).

```
from collections import deque
file = deque([1, 2, 3])      # 1 est en tête de file, 3 est en fin de file
file.append(4)               # deque([1, 2, 3, 4])
x = file.popleft()           # x == 1  et file == deque([2, 3, 4])
```

Remarques :

- On pourrait aussi utiliser **appendleft()** et **pop()**.
- Il existe aussi **extend()** et **extendleft()** qui ajoutent plusieurs éléments (itérable)

Voir la documentation de deque, généralisation des files et des piles.

Remarque : deque est préférable à list pour réaliser un pile.

Ensemble (set) : équivalent aux ensembles en math.

- Pas de numéro d'ordre sur les élements (et donc pas d'accès par indice)
- Pas de double : ajouter un élément déjà présent ne change pas l'ensemble
- **Principales opérations :** ajoute un élément, supprime un élément, appartenance, taille
- Relation d'ordre partielle : inclusion

```
s1 = {1, 2, 3, 1, 2}     # s1 == {1, 2, 3}          # Pas de double !
s2 = set([5, 4, 3])      # à partir d'une liste (un itérable), ordre sans importance
s3 = {3, 4}
s4 = set()               # Un ensemble vide
```

op.	méthode	exemple
in		1 in s1 is True
	add(x)	ajoute l'élément dans l'ensemble
	pop()	supprime et retourne un élément au hasard
	discard(x)	supprime x de l'ensemble, rien si x not in set
	remove(x)	supprime x ou lève KeyError si x not in set
<=	issubset(other)	s3 <= s2 is True (ou s3.issubset(s2))
>=	issuperset(other)	s1 >= s2 is False and s2 >= s1 is False
|	union(other)	s1 | s2 == {1, 2, 3, 4, 5}
&	intersection(other)	s1 & s2 == {3}
-	difference(other)	s1 - s2 == {1, 2}
^	symmetric_difference(other)	s1 ^ s2 == {1, 2, 4, 5}
|=	update(x)	s1.update(s2) ; s1 == {1, 2, 3, 4, 5}
	clear()	vide l'ensemble

Remarque : frozenset est un ensemble immuable

Dictionnaire (dict)

- **Définition :** Permet d'utiliser une clé (key) pour enregistrer et récupérer une information.
- **Remarque :** Un genre de généralisation des listes où l'indice devient n'importe quel type.
- **Principales opérations :** ajouter une valeur avec sa clé, récupérer une valeur grâce à une clé
- **Synonyme :** Tableaux associatifs
- Pas de numéro d'ordre sur les éléments (et donc pas d'accès par indice entier)
- **Attention :** La clé doit être hashable

```
d1 = {}                          # ou d1 = dict() : un dictionnaire vide
d1 = {'I': 1, 'X': 10, 5: 'V'}   # dictionnaire avec des couples clé:valeur
d2 = dict(un=1, deux=2)          # d2 == {'un': 1, 'deux': 2}, les clés sont des chaînes

print(d1['I'], d1[5])            # 1 V                                          (accès)
d1['I'] = 'A'                    # d1 == {'I': 'A', 'X': 10, 5: 'V'}          (modification)
del d1['I']                      # d1 == {'X': 10, 5: 'V'}                     (suppression)
x = d2.pop('un')                 # supprime la clé 'un' et retourne sa valeur (car 'un' in d2)
x = d1.pop('I', 'oups!')         # d1 inchangé et retourne 'oups!' (car 'I' not in d1) pop(clé [,defaut])
'I' in d1                        # False : est-ce que 'un' est une clé de d1 ?
x = d1.get('I', 'oups!')         # x == 'oups!' car x not in d1.                get(clé [, defaut])
i = d1.items()                   # dict_items([('X', 10), (5, 'V')]) # ensemble des couples (clé, valeur)
k = d1.keys()                    # dict_keys(['X', 5])               # ensemble des clés
v = d1.values()                  # dict_values([10, 'V'])            # conteneur des valeurs
# Remarque : items(), keys(), values() sont des vues sur le dictionnaire.
d1[10] = 'A'                     # d1 == {'X': 10, 5: 'V', 10: 'A'}
print(k, v)                      # dict_keys(['X', 5, 10]) dict_values([10, 'V', 'A'])
d1.update({'X': 'A', 'L': 50}) # d1 == {'X': 'A', 5: 'V', 'L': 50}
x = d1.setdefault('X', 10)       # x == 'A'     # d1 inchangé car 'X' in d1
x = d1.setdefault('C', 100)      # x == 100     # et {'C' : 100} ajouté dans d1
d1.setdefault([1], 'ok?')        # TypeError: unhashable type: 'list'
```

Exercices

1. Quelle est la structure de données adaptée pour représenter :
   - un classeur
   - un cahier
   - un répertoire téléphonique
   - les couleurs possibles pour une pomme
   - les prénoms d'une personne
   - les membres d'une équipe de projet
2. Comment obtenir le nombre de caractères différents utilisés dans un texte ?
3. Quelle structure de données utiliser pour représenter la note des étudiants sur une épreuve ?
   a. Pierre a 15, Julie 18, Jean 8, Zoé 15. Comment le représenter ?
   b. Comment corriger la note de Jean : il a eu 9.
   c. Comment obtenir la meilleure note ?
   d. Combien de notes différentes ont été attribuées ?
   e. Comment afficher les notes sous la forme : « Prénom : Note » ?
   f. Comment savoir si un étudiant a passé l'épreuve ?
   g. Comment obtenir la note d'un étudiant (on suppose qu'il a eu 0 s'il n'a pas passé l'épreuve) ?
   h. Est-ce une bonne idée d'utiliser le prénom ?
4. Comment obtenir la fréquence (le nombre d'occurrences) des caractères d'un texte ?

Sommaire

Les fonctions comme données

Les fonctions sont des objets

```
def f1():
    print("C'est f1 !")

type(f1)     # <class 'function'>
f2 = f1      # Un autre nom donne accès à la fonction attachée à f1
f2()         # affiche "C'est f1 !"
f2.__name__ # 'f1'

def g(f):    # une fonction f en paramètre
    print('début de g')
    f()      # f doit être « appelable » (callable)
    print('fin de g')

g(f1)        # "début de g" puis "C'est f1 !" puis "fin de g"
```

Les lambdas : fonctions courtes, anonymes, avec une seule expression

```
cube = lambda x : x ** 3     # à éviter

def cube(x):                 # Mieux !  Plus clair !
    return x ** 3
```

Calcul d'un zéro d'une fonction continue

```
def zero(f, a, b, *, precision=10e-5):
    '''Retourner une abscisse où la fonction f s'annule entre a et b'''
    assert f(a) * f(b) <= 0  # et f doit être continue...
    if a > b:
        a, b = b, a
    while b - a > precision:    # par dichotomie
        milieu = (a + b) / 2
        if f(a) * f(milieu) > 0:
            a = milieu
        else:
            b = milieu
    return (a + b) / 2

def equation1(x):
    return x ** 2 - 2 * x - 15

x = zero(equation1, 0, 15)
assert abs(5 - x) <= 10e-5

x = zero(lambda x : 2 * x - 5, -5, 10)     # lisible ?
import math
assert math.isclose(x, 2.5, rel_tol=1e-4)
```

Autre exemple : trier

Python propose une fonction prédéfinie `sorted` qui permet d'obtenir une liste triée à partir d'une collection (en fait un itérable).

```
>>> s = [1, 4, -18, 5, 2, 3]
>>> sorted(s)
[-18, 1, 2, 3, 4, 5]
```

Elle prend un paramètre optionnel `key` pour définir la clef de tri (chaque élément sera trié suivant la valeur que retourne `key` sur lui).

```
>>> sorted(s, key=abs)                # sur la valeur absolue des nombres
[1, 2, 3, 4, 5, -18]
>>> sorted(s, key=lambda x : -x)      # en sens inverse
[5, 4, 3, 2, 1, -18]
>>> sorted(s, reverse=True)           # ou en utilisant l'option reverse
[5, 4, 3, 2, 1, -18]
>>> sorted(s, key=lambda x : x % 2) # les pairs avant les impairs
[4, -18, 2, 1, 5, 3]
```

Remarque : Il existe une méthode `sort` sur `list` qui trie les éléments sur place.

```
>>> s.sort(key=lambda x : x % 2, reverse=True)      # retourne None
>>> s
[1, 5, 3, 4, -18, 2]
```