Retour sur les fonctions

Un peu d'histoire

Ada Lovelace, née Ada Byron le 10 décembre 1815 à Londres et morte le 27 novembre 1852 à Marylebone, est une pionnière de la science informatique.

Elle est principalement connue pour avoir réalisé le premier véritable programme informatique, lors de son travail sur un ancêtre de l'ordinateur : la machine analytique de Charles Babbage.

Automatismes

On note `L = [12, 3, 7, 1]`.
Que renvoie `sum(L)` ?

On note `c = "Numérique"`.
Que renvoie `len(c)` ?

On note `t = [5, 1, 2]`.
Que renvoie `t.append(7)` ?

Exercice 1 : Pour commencer

On considère la fonction écrite en python :

```
def test(a, b):
    if a < b:
        res = b
    else:
        res = a
    return res
```

1. Quelle valeur produit l'instruction `test(12, 5)` ? et `test(test(1, 2), test(4, 3))` ?

2. a. On rappelle que la fonction `abs` correspond à la valeur absolue définie par :

$$\text{abs}(x) = \begin{cases} x & \text{si } x \geqslant 0 \\ -x & \text{si } x < 0 \end{cases}.$$

Démontrer que si $a \geqslant b$, alors $\text{abs}(a-b) = a-b$.

b. On considère la fonction définie par :

```
def test2(a, b):
    return (a+b+abs(a-b))/2
```

i. Que renvoie l'instruction `test2(5, 3)` ?

ii. Vérifier que la fonction `test2` a le même rôle que la fonction `test`.

3. Que renvoie les instructions `test("N", "S")` et `test2("N", "S")` ?

D.W.

Cours

Une version de la fonction `test` précédente qui respecte les bonnes pratiques de programmation en python est donnée par :

```
def maximum_entre_deux_nombres(a:float, b:float ) -> float:
    """ Renvoie le maximum entre deux nombres. """
    return (a+b+abs(a-b))/2
```

On y retrouve :

- ❏ Un nom explicite écrit en `snake_case` ;
- ❏ Une liste d'arguments où on précise les types attendus ;
- ❏ Le type d'élément que renvoie la fonction ;
- ❏ Une **docstring** décrivant la fonction ;
- ❏ Le mot clé `return` pour indiquer la fin de la fonction et ce qu'elle renvoie.

Attention : il ne peut y avoir qu'un seul `return` par fonction.

Exercice 2 : Décomposer en sous-problèmes

On cherche dans cet exercice à tester si une date fournie en paramètre est valide ou non.

1. a. Dire si chacune des dates ci-dessous est valide ou non :

22/7/1987	25/14/1970	-13/10/1938
16/5/-450	31/6/2024	29/2/2004

b. Lister les différentes contraintes que doit respecter une date.

2. La vérification d'une date peut se décomposer en deux sous-problèmes :

❏ L'année est-elle bissextile ? ❏ Combien y a-t-il de jours dans le mois demandé ?

On suppose que les fonctions python `est_bissextile(annee:int) -> boolean` et `jours_dans_mois(mois:int, annee:int) -> int` ont été écrites.

a. Que renvoie `jour_dans_mois(7, 2024)` et `est_bissextile(2024)` ?

b. Compléter le code de la fonction `date_valide(j:int, m:int, a:int) -> boolean` ci-dessous :

```
1  def date_valide(j:int, m:int, a:int) -> boolean:
2      """ Vérifie si une date est valide. """
3      if not 1 <= m <= 12:
4          res = False
5      else:
6          jours = jours_dans_mois(..., ...)
7          if 1 <= j <= jours:
8              res = ...
9          else:
10             res = ...
11     return res
```

3. Proposer un code pour la fonction `jour_dans_mois`.
Par exemple, l'instruction `jour_dans_mois(2, 2024)` renvoie 29.

Cours

Lorsqu'une fonction renvoie un booléen (`True` ou `False`), il est important d'être concis dans l'écriture du code. Par exemple, pour tester si un nombre est divisible par 5, on pourrait écrire naïvement :

```
def est_divisible_par_5(n:int) -> boolean:
    """ Teste si un nombre est divisible par 5 en regardant ses unités. """
    if n%10 == 0 or n%10 == 5:
        res = True
    else:
        res = False
    return res
```

qui pourrait se lire : « si cette condition est vraie alors on renvoie vrai, sinon on renvoie faux. ».
On peut raccourcir cette fonction par :

```
def est_divisible_par_5(n:int) -> boolean:
    """ Teste si un nombre est divisible par 5 en regardant ses unités. """
    return n%10 == 0 or n%10 == 5:
```

Attention à bien se souvenir des opérateurs booléens `and`, `or` et `not`.

Exercice 3 : Retour sur les expressions booléennes

Donner la valeur booléenne de chacune des expressions ci-dessous :

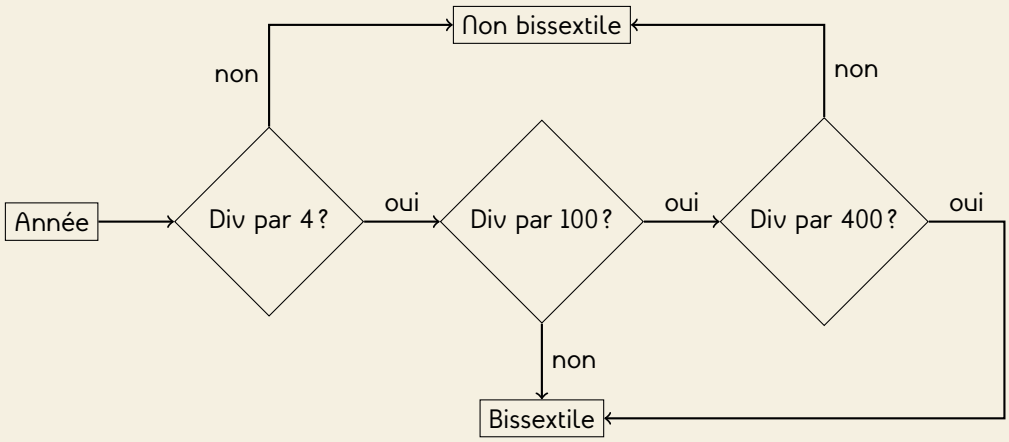
- ❑ `(3 in [6, 2, 4, 3, 5]) and (13 < 8)`
- ❑ `((not 18 != 5) or ("S" in "NSI")) and (2024//10 == 4)`
- ❑ `not((len("NSI") == 3) and ("NSI" < "CHIMIE")) or est_bissextile(2000)`

Exercice 4 : Application aux années bissextiles

On commence par rappeler la règle fondamentale. Une année est bissextile si :

❑ Elle est divisible par 4 mais pas par 100 ou si elle est divisible par 400.

On peut résumer la règle par le graphe décisionnel suivant :



1. Tester cette règle sur les années 2024, 2026, 2000 et 1900.
2. Écrire une fonction `est_bissextile(a:int) -> boolean` correspondant à cette règle.
Attention, la fonction doit être concise.

Exercice 5 : Les nombres parfaits

On rappelle qu'un nombre est **parfait** s'il est égal à la moitié de la somme de ses diviseurs.

Un peu d'histoire

Les quatre premiers nombres parfaits sont connus depuis l'Antiquité. On les retrouve dans les travaux de Nicomache de Gérase et de Théon de Smyrne. Le cinquième nombre parfait est mentionné dans un codex latin de 1456. Les sixième et septième nombres parfaits ont été trouvés par Cataldi au XVIème siècle et le huitième en 1772 par Euler.

1. Vérifier que 6 est un nombre parfait mais que 24 ne l'est pas.

On décompose le problème en trois sous-problèmes :

Sous-problème	Fonction python associée
❑ Trouver tous les diviseurs d'un entier	`diviseurs(n:int) -> list`
❑ Sommer les éléments d'une liste	`somme_liste(L:list) -> int`
❑ Vérifier si un nombre est parfait	`est_parfait(n:int) -> boolean`

2. On suppose que `diviseurs` et `somme_liste` ont été écrites.
Compléter le code de la fonction `est_parfait`.

```
def est_parfait(n:int) -> boolean:
    """ Teste si un nombre est parfait. """
    return n == ... // 2
```

Cours

Il existe deux façons de parcourir une liste à l'aide d'une boucle bornée `for`.

En parcourant les indices	En parcourant les éléments
`for i in range(len(L)):` `  ## Traitements sur L[i]`	`for elt in L:` `  ## Traitements sur elt`
Pour `L = [8, 4, 1, 7]`, la variable `i` vaut successivement 0, 1, 2 et 3.	Pour `L = [8, 4, 1, 7]`, la variable `elt` vaut successivement 8, 4, 1 et 7.

Attention à bien s'interroger sur l'utilité du choix effectué.

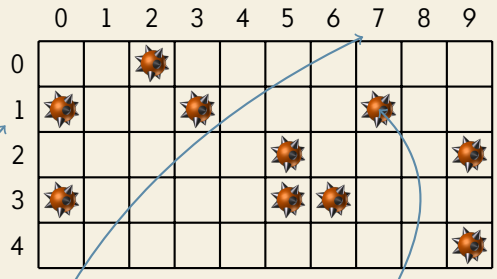
3. Pour déterminer les diviseurs d'un entier `n`, on doit tester si chacun des entiers de `1` à `n` divise `n`.
Compléter le code de la fonction `diviseurs`.

```
def diviseurs(n:int) -> list:
    """ Renvoie la liste des diviseurs de n. """
    res = []
    for ... : # i varie de 1 à n
        if ... : # Teste si i divise n
            res.append(...) # Ajoute i à la liste res
    return res
```

4. Écrire le code de la fonction `somme_liste(L:list) -> int`.
Par exemple, l'instruction `somme_liste([5, 1, 8, 12])` renvoie `26`.

Exercice 6 :

On possède une grille de démineur avec 5 lignes et 10 colonnes que l'on souhaite remplir avec un certain nombre de bombes placées aléatoirement.



On modélise cette grille à l'aide d'une liste de listes où les cases vides sont associées à `0` et les bombes à `-1`. On obtient ainsi :

```
grille = [[ 0, 0, -1,  0, 0,  0,  0,  0, 0,  0],
          [-1, 0,  0, -1, 0,  0,  0, -1, 0,  0],
          [ 0, 0,  0,  0, 0, -1,  0,  0, 0, -1],
          [-1, 0,  0,  0, 0, -1, -1,  0, 0,  0],
          [ 0, 0,  0,  0, 0,  0,  0,  0, 0, -1]]
```

L'instruction `grille[ 1 ][ 7 ]` correspond à la bombe située ligne 1 et colonne 7.

1. Écrire une fonction `creer_grille(n:int, m:int) -> list` qui renvoie une liste modélisant une grille de démineur à `n` lignes et `m` colonnes ne contenant aucune bombe.

Cours

Quand on place aléatoirement une bombe dans une grille, il se peut que la place soit déjà occupée par une autre bombe. Ainsi, on ne sait pas à l'avance combien de tentatives de placement on doit réaliser.

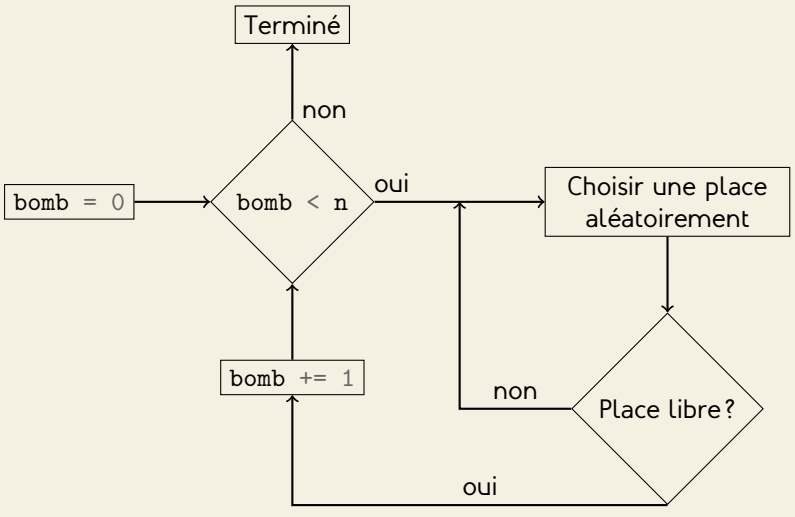
Il faut alors utiliser une **boucle non bornée**. Une telle boucle se réalise suivant la structure :

```
while (condition):
    # traitements
```

Attention à ce que la condition devienne fausse un jour pour pouvoir sortir de la boucle.

2. a. Écrire une fonction `taille_grille(g) -> tuple` qui renvoie la taille de la grille `g` sous la forme `(ligne, colonne)`.

b. Écrire à l'aide de l'algorigramme ci-dessous, la fonction `remplir_grille(g:list, n:int) -> None` qui place aléatoirement `n` bombes dans la grille `g`.



D.W.