TP1

Partie 1 — Expressions

Ex.1 — Interpréteur et arithmétique

Dans ce cours vous apprendrez le langage Python : un langage moderne, facile d'accès et très utilisé. Vous travaillerez dans pycharm : une IDE (*Integrated Development Environment*) pour Python, c'est-à-dire un éditeur spécialisé dans la rédaction de programmes python, leur exécution, débogage, etc.

Question 1 : Parcourir rapidement Mettre la Documentation Python 3 dans vos signets. Faire le Quick tour.

Note : si vous ne comprenez pas l'anglais… il faut s'y mettre, il est omniprésent dans le monde de la programmation, de l'ingénierie et des Sciences en général.

Python est un langage interprété : pas besoin, donc, d'écrire le programme complet et de le compiler pour pouvoir enfin l'exécuter; on peut écrire et faire exécuter du python ligne par ligne.

Question 2 : Ouvrir la console python en bas à gauche de pycharm. Ecrire

```
>>> 2+2
```

puis entrée.

Noter que `2+2` fait partie du langage python. En appuyant sur entrée vous transmettez ce fragment de code python à l'interpréteur, qui l'évalue et renvoie 4.

Il existe probablement plusieurs versions de python sur votre machine, il faut s'assurez que vous utilisez la bonne.

Question 3 : Lorsque vous avez ouvert la console, à la question précédente, la première chaîne de caractère qui s'est affiché indiquait le chemin d'accès de l'interpréteur python que vous utilisez. Assurez vous qu'il s'agisse bien du Python 3 fourni par Anaconda3. Sinon revoir a fin de *AMETICE > L1MEO > Pour lancer vos TPs* pour arranger ça, et relancer la console.

Question 4 Exécuter dans la console :

```
>>> 7+3*5
>>> (7+3)*5
>>> 20/7
>>> 20//7
>>> 20 % 7
>>> 2,5/5
>>> 2.5/5
```

Noter les priorités entre opérateurs: `*` a précédence sur `+`.

Noter que `/` est la division, `//` est la division entière, et `%` est le reste de la division entière.

Noter que le séparateur décimal est le point, et non pas la virgule, car celle-ci est déjà utilisée pour séparer les éléments d'une liste.

Correction

```
# TODO : Utiliser le shell et la console, en cliquant en bas à gauche, pour cet exercice.
```

Ex. 2 — Variables and types

Variables

En python les variables sont comme des étiquettes que l'on pose sur les objets, pour leur donner un nom.

Un nom de variable est : * une séquence de lettres (a → z , A → Z) et de chiffres (0 → 9), qui commence par une lettre. * Les lettres accentuées, les cédilles, les espaces, les caractères spéciaux etc. sont interdits, à l'exception de l'underscore _. * Les lettres majuscules et minuscules sont distingués: `Joseph`, `joseph`, `JOSEPH` sont donc des variables différentes. * Choisissez des noms de variables clairs, significatifs.

L'instruction python

```
code=4287
```

pose l'étiquette `code` sur l'objet `4287`. Désormais, `code` se verra affectée la valeur `4287`. Cette instruction est une *affectation*. On décide que `code` vaut `4287`, c'est une affirmation, pas une question.

A l'inverse, l'instruction python

```
code==4287
```

interroge l'interpréteur sur le fait que `code` vaille `4287`, ou pas. La réponse peut être `True` (vrai) ou `False` (faux). Cette instruction de modifie pas la valeur de `code`. C'est une question, pas une affirmation.

Question 1 : Affecter les valeurs `7.692` et `21.8` aux variables `a` et `b`. Calculer leur somme dans une variable nommée `resultat` et affichez le contenu de cette dernière.

Question 2 : On cherche à calculer la valeur d'un téléphone valant 999 USD en euros. Affecter cette valeur à une variable `telephone_usd`. En supposant que 1 USD = 0.9 EUR, calculer la conversion dans une variable

nommée `telephone_eur`. En utilisant les flèches haut-bas, changer la valeur de `telephone_usd` à 888 USD et recalculer `telephone_eur`.

Types

Combien valent 3 chaussettes plus 4 caleçons?

La question est déconcertante, peut-être est-elle mal posée, il est parfois difficile de mélanger ainsi des objets de différents types.

En python, les objets ont des types : * **int** : désigne un entier * **float** : désigne un nombre décimal * **str** : désigne une chaîne de caractères On peut consulter le type d'un objet à l'aide de la fonction `type(.)`.

Question 3 Exécuter, dans la console:

```
>>> type(2)
>>> a=2
>>> type(a)
>>> type(10 // 3)
>>> type(10 / 3)
>>> type(2.5)
>>> a=2.5
>>> type(a)
>>> type(2 + 2.5)
>>> type(3 * 1.0)
>>> type(3.14 * 1)
>>> type("Bonjour")
>>> type("""Aurevoir""")
>>> type('!!!')
>>> type("2")
>>> a="2"
>>> type(a)
```

Noter que les variables ne sont pas d'un type figé en python, car ce sont simplement des étiquettes que l'on pose sur des objets, qui eux ont un type figé.

Noter que les trois types de guillemets qui permettent de former des chaînes de caractère en python.

Noter que python accepte de faire des opérations mélangeant `int` et `float`, et que cela donne un objet de type `float`. Cela a du sens.

Par contre, additionner des nombres à des chaînes de caractère n'a pas de sens, et python ne vous signale alors une erreur.

Question 4 Exécuter, dans la console:

```
>>> 2+"hello" #Erreur de type
>>> 2+"2" #Erreur de type
```

```
>>> nombre=2
>>> mot="hello"
>>> nombre+mot #Erreur de type
>>> nombre+nombre
>>> mot+mot
```

Remarquer que `+` est, tantôt une opération sur les nombres, tantôt une opération sur les chaînes de caractères, mais n'accepte pas de mélange des deux.

Remarquer en effet que `2` n'a rien à voir avec `"2"` car l'un est un entier, tandis que l'autre est une chaîne de caractères.

Toutefois, l'un peut être converti en l'autre si on le souhaite vraiment:

Question 5 Exécuter, dans la console:

```
>>> 2+int("2")
>>> str(2)+"2"
```

Question 5 Exécuter, dans la console:

```
>>> x=2
>>> x+x
>>> x="hello"
>>> x+x
```

Remarquer que python ne peut pas savoir, en voyant juste la ligne `x+x`, si l'on est en train de lui demander de faire le `+` sur les entiers ou bien le `+` sur les chaînes de caractères. Il a besoin pour ça de connaître le type de `x`, mais celui-ci peut varier au cours du programme. Le comportement de `x+x` peut donc varier aussi au cours du programme; le terme technique pour désigner ceci est le "typage dynamique".

Correction

```
# Cet exercice se fait dans la console python.
```

Ex. 3 — Manipulations & affichages

Question 1 : Exécuter dans la console :

```
>>> print("hello")
>>> mot="hello"
>>> print(mot)
>>> print(mot+mot)
>>> print(mot+" "+mot)
```

```
>>> print(mot,mot)
>>> print(2)
>>> nombre=2
>>> print(nombre)
>>> print(nombre+nombre)
>>> print(nombre,nombre)
>>> print(mot+nombre) #Erreur de type
>>> print(mot,nombre)
```

Question 2 : * Définir deux variables : `a` et `b` ayant pour valeurs respectivement `5` et `15`. * En une ligne de code, afficher la phrase : "`a vaut 5 et b  vaut 15, leur somme fait 20`"—sans tricher: en utilisant les valeurs des variables à ce moment là. * Changer `b` en `10` et réexécuter la ligne, pour voir si l'affirmation de la phrase reste correcte.

Question 3 : * En utilisant l'opérateur `%`, et en une ligne de code, afficher la phrase : "`That b is a multiple of a is True`"—sans tricher: en utilisant les valeurs des variables à ce moment là. * Changer `b` en `13` et réexécuter la ligne, pour voir si l'affirmation de la phrase reste correcte.

Question 4 : Écrivez les lignes de code permettant d'échanger les valeurs de `a` et de `b`. Sans tricher: vos lignes de codes ne doivent pas dépendre des valeurs spécifique que contiennent `a` et `b` initialement. Réexécuter la ligne de la question 2 pour voir si ça a marché.

Question 5 : A la question 4, est-ce que vous avez eu besoin d'une troisième variable? Ca n'était pas indispensable. Utilisez, soit votre esprit mathématique, soit votre esprit geek, pour vous en dispenser.

Correction

```
# Cet exercice se fait dans la console python.
```

Partie 2 — Programmes

Ex. 4

Un programme est une combinaison d'instructions, tout comme dans une recette de cuisine. Les blocs instructions qui se suivent forment une séquence; l'interpreteur prendra ce bloc et les exécutera l'une après l'autre.

Vous écrirez vos programme dans le fichier `task.py`, dans le pannel du milieu.

"L'état normal d'un programme, c'est de bugger." —Gérard Berry, Professeur d'informatique au Collège de France.

Le remède, c'est de tester votre programme à chaque fois que vous le modifiez. Pour ça, clique droit sur `task.py`, et `Run`.

Question 1 : Écrire un programme qui affecte les variables `note1`, `note2`, `note3` avec les valeurs `15.5`, `12.75` et `14.25` respectivement. Ensuite, calculer la moyenne de ces notes et stocker le résultat dans une variable nommée `moyenne`. Affichez la valeur de la variable `moyenne` précédée du message : "La moyenne des trois notes est".

Question 2 : Si 15 étudiants utilisent 5 ordinateurs, combien utiliseront d'ordinateurs 90 étudiants ? Écrire un programme contenant des variables bien nommées avec les données du problème, puis calculant le résultat dans une variable et l'affichant.

Quand vous aurez terminé, appuyer sur le bouton `Check` du pannel de droite. Cela vous indiquera si votre réponse est exactement celle que nous attendions. Si votre réponse diffère mais qu'elle marche et reste élégante, vous pouvez la conserver.

Une fois le bouton `Check` appuyé, si vous bloquez, vous pouvez consulter la solution en appuyant sur Peek…

Correction

```
#Cet exercice se fait en modifiant ce fichier.
#1. Ne modifiez que l'intérieur les zones marquées
#   (les "placeholders").
#2. Exécutez votre programme à chaque fois que vous y ajoutez une ligne de code !!!
# Pour ça :
#   - Clique droit sur task.py
#   - Run
#   - La fenêtre d'exécution remplace la console en bas.
#3. Quand vous aurez terminé, appuyer sur le bouton Check du pannel de droite.
# Cela vous indiquera si votre réponse est exactement celle que nous attendions.
#4. Une fois le bouton Check appuyé, si vous bloquez, vous pouvez consulter la solution en a

#Q1
note1 = 15.5
note2 = 12.75
note3 = 14.25
moyenne = (note1 + note2 + note3) / 3
print("La moyenne des trois notes est : ", moyenne)

#Q2
etudiants = 15
ordinateurs = 5
```

```
ratio = ordinateurs / etudiants
resultat = 90 * ratio
print(resultat)
```

TP2

Partie 1 — Les expressions booléennes

Ex. 1

Cet exercice est à effectuer dans la console Python.

Question 1 : Exécuter dans la console:

```
>>> True
>>> 1+1==2
>>> False
>>> 1+1==3
>>> type(True)
>>> type(False)
>>> type(1+1==2)
```

Noter que `True` *et* `False` *sont des valeurs en python, au même titre que* `1, 2, 3...`. *Par contre* `True` *et* `False` *sont des valeurs de type* `bool` *et non pas* `int`.

De même que l'on peut faire des expressions arithmétiques qui calculent sur les entiers, ont peut faire des expressions booléennes qui calculent sur les booléens.

Les opérateurs booléens sont : * `and` : "et" * `or` : "ou" * `not` : "non"

Question 2 : Exécuter dans la console:

```
>>> True and False
>>> True and True
>>> True or False
>>> True or True
>>> True and (True or False)
>>> not False
>>> not True
```

Question 3 : Exécuter dans la console:

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

Réévaluer cette dernière expression booléenne avec les trois autres combinaisons de valeurs pour `a` et `b`.

En logique, cette égalité est connue sous le nom de Loi de De Morgan.

On peut bien sûr générer des valeurs booléennes en comparant des valeurs non-booléennes. Les opérateurs de comparaison sont:

- `<` : "strictement inferieur à"
- `<=` : "inférieur ou égal à"
- `>` : "strictement supérieur à"
- `>=` : "supérieur ou égal à"
- `==` : "égal à"
- `!=` : "différent de"

Attention: `==` correspond à une question : c'est deux valeurs sont-elles égales? Tandis que `=` correspond à une affectation : on décide qu'une certaine variable doit désormais valoir une certaine valeur.

Question 4 : Exécuter dans la console:

```
>>> 3 == 4
>>> 3 != 4
>>> 3 <= 4
>>> 3+4 >= 7
>>> 3+4==7 or 3<=7-5
>>> 24 % 2 == 0 and 24 % 3==0
```

Question 5 : Initialiser les variables a, b et c respectivement aux valeurs 2, 5 et 7. Déterminer si les tests suivants sont vrais ou faux en utilisant les opérateurs booléens et les opérateurs de comparaison correspondants : * a **égale à** b * (a + b) **différent de** c * **non** (a **égale à** c) * ((a + b) **supérieur ou égal à** c) **et** (a **inférieur ou égal à** (c − b)) * ((a + c) > b) **ou** ((a + c) < b)

Correction

```
# Cet exercice est à effectuer dans la console Python.
```

Partie 2 — If

Ex. 2

L'instruction `if` signifie "si". Elle permet de n'exécuter un bloc d'instruction que si une certaine condition (une expression booléenne) s'évalue à vrai.

```
if expr :
    # bloc d'instructions
    # qui n'est exécuté que
    # que si expr est vraie
# suite des instructions
```

IMPORTANT : Il y a une indentation de 4 espaces (ou une tabulation) entre la ligne du `if`, et les lignes du bloc. Ce type d'espace laissé dans la marge de gauche s'appelle l'indentation. L'indentation est primordiale en python, car elle permet de voir quand commence et s'arrête le bloc d'instruction qui est conditionné par le `if`.

Les deux points après la condition sont indispensables aussi. Ils signifient "alors". Par exemple, la phrase française "si x est pair alors diviser x par deux" se traduit en python par:

```
if x % 2 == 0 :
    x = x // 2
```

Variation avec alternative:

```
if expr :
    # bloc d'instructions
    # qui n'est exécuté que
    # que si expr est vraie
else :
    # bloc d'instructions
    # qui n'est exécuté que
    # que si expr est fausse
# suite des instructions
```

Variation avec cas multiples:

3. Cas multiples

```
if expr1 :
    # bloc d'instructions
    # qui n'est exécuté que
    # que si expr1 est vraie
elif expr2:
    # bloc d'instructions
    # qui n'est exécuté que
    # que si expr2 est vraie
else :
    # bloc d'instructions
    # qui n'est exécuté que
    # que si expr1 et expr2 sont fausses
# suite des instructions
```

Question 1 : Affecter la valeur `17` à la variable `note`. Ecrire deux lignes de

code qui correspondent à la phrase française “si la valeur de a est supérieure ou égal à 17 alors afficher bravo!”. Tester vos deux lignes en changeant la valeur de `a`.

L'instruction `input(.)` sert à poser une question à *l'utilisateur* du programme (à ne pas confondre avec le programmeur du programme!). * A l'intérieur de la parenthèse on met la question que l'on veut poser. Elle s'affichera comme si c'était un `print(.)`. * L'expression entière prendre comme valeur la chaîne de caractère qu'aura saisi l'utilisateur en réponse à votre question.

Par exemple

```
rep=input("Comment allez-vous?")
```

Affiche “Comment allez-vous?”; attend de l'ulisateur qu'il tape une réponse; et met cette réponse dans `rep`.

Question 2 : Ecrivez un programme python qui pose la question “Bonjour, comment ça va?”. Si l'utilisateur répond “Bien” alors affichez le message “Super! Je suis content pour vous.” Si la réponse est différente de “Bien” affichez “Mince… courage!”.

Question 3 : Recopier et modifier la question 1 pour que le programme demande à l'utilisateur de saisir sa note, avant de lui dire si celle-ci est au-dessus de la moyenne ou pas.

Noter qu'il vous faut convertir le résultat de `input(.)` en entier avec la fonction `int(.)` afin de pouvoir le comparer à 10.

Question 4 : Ecrire un programme qui demande son âge à l'utilisateur : “Quel est votre âge ?” et s'il oui ou non il souhaite des pop corns : “Souhaitez-vous des pop corns ?”. Le programme affiche alors le `prix` de sa séance de cinéma. Si la personne est mineure, le prix d'une place est de 7EUR, si la personne est majeure le prix est de 12EUR. Le pop corn est à 5EUR.

Correction

```
#Q1
note = 17
if note >= 10:
    print("Vous avez la moyenne")

#Q2
rep = input("Bonjour, comment allez-vous?")
if rep == "Bien":
    print("Super! Je suis content pour vous.")
else:
```

```
    print("Mince... courage!")

#Q3
note = int(input("Quelle est votre note? "))
if note >= 10:
    print("Vous avez la moyenne")

#Q4
age = int(input("Quel est votre age ?"))
if age < 18:
    prix= 7
else:
    prix= 12
if input("Souhaitez-vous un pop corn ?")=="oui" :
    prix+= 5
print("Le prix total est de :",prix)
```

Partie 3 — For

Ex. 3

L'instruction `for` signifie "pour". Elle permet de répéter l'exécution d'un bloc d'instruction un nombre fixe de fois; tout en faisant varier la valeur d'une certaine variable.

```
for i in range(initial, final) :
    # bloc d'instructions
    # qui s'exécute pour `i` allant de
    # la valeur initiale incluse
    # à la valeur finale excluse
# suite des instructions
```

Par exemple, la phrase française "pour i allant de 1 à 5 affiche Hello!" se traduit en python par

```
for i in range(1,6) :
    print("Hello!")
```

Pour faire un "countdown" de fusée, on pourrait utiliser

```
for i in range(10,-1,-1) :
    print(i,"...")
print("Go !!!")
```

On peut aussi faire varier `i` dans une liste arbitraire.

```
print("Les premiers nombres premiers sont ")
for i in [2,3,5,7] :
```

```
    print(i)
```

D'ailleurs, nul besoin d'utiliser spécifiquement la variable `i`.

Question 1 : Ecrire un programme qui calcule et affiche les cubes de 1 à 100. Remarque: utiliser l'opérateur `**` pour la puissance.

Question 2 : Ecrire un programme qui calcule la moyenne des cubes de 1 à 100, et affiche le résultat final.

Question 3 : Ecrire un programme qui calcule la somme des nombres de la forme $1/2^i$, pour i allant de 1 à 100, et affiche le résultat final.

Correction

```
start=1
end=100

#Q1
for i in range(start,end+1):
    print(i**3)

#Q2
somme=0
for i in range(start, end+1):
    somme += i**3
print(somme /(end+1-start))

#Q3
somme=0
for i in range(start, end+1):
    somme += 1/2**i
print(somme)
```

Ex. 4 — Fibonacci

Écrire un programme qui affiche successivement les 20 premiers nombres de la suite de Fibonacci de 20. En termes mathématiques cette suite est définie par:

- $F(0) = 0$, $F(1) = 1$
- $F(n) = F(n-2) + F(n-1)$ pour $n > 1$

La difficulté, ici, est que nous ne savons par encore comment définir des fonctions du style $F(n)$, ni même des listes du style F_n. Nous apprendrons ça, mais pour

l'heure, nous allons nous en passer.

Au lieu de ça nous utiliserons seulement trois variables: `Fn_moins_2` initialisée à `0`,`Fn_moins_1` initilisée à `1`, et `Fn`.

Nous afficherons les valeurs se la suite en procédant ainsi:

Afficher `0`

Afficher `1`

Pour les 18 prochains termes:

- calculer `Fn=Fn_moins_1+Fn_moins_2` et l'afficher.
- Mettre `Fn_moins_1` dans `Fn_moins_2`.
- Mettre `Fn` dans `Fn_moins_1`.

Question 1 : Ecrire le programme en question.

Correction

```
#Q1
Fn_moins_2=0
Fn_moins_1=1
print(0)
print(1)
for i in range(1,19):
    Fn=Fn_moins_1+Fn_moins_2
    print(Fn)
    Fn_moins_2=Fn_moins_1
    Fn_moins_1=Fn
```

Ex. 5 — Pyramides

Si je répète 5 fois, le fait de répéter 10 fois "Hello!", combien de fois est-ce que je répète "Hello!" au total?

Une boucle imbriquée est une boucle inclue dans une autre. Cela correspond à l'idée de répéter (boucle extérieure) une répétition (boucle intérieure).

Voici un example.

```
num_lins = 10
num_cols = 5
for i in range(num_lins):
    for j in range(num_cols):
```

```
        print('*', end="")
    print("")
```

Affichage après exécution :

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

Remarquer le ***print("")*** *qui affiche la chaîne vide… mais affiche aussi un saut de ligne, car tout* ***print(.)*** *termine son affichage par un saut de ligne. Sauf si on lui demande de ne pas le faire, en spécifiant* ***print(.,end="").***

Question : Écrire un programme qui affiche à l'écran une pyramide remplie d'étoiles, sur le modèle ci-dessous. La dernière ligne contient 10 étoiles

```
*
* *
* * *
* * * *
* * * * *
* * * * * *
* * * * * * *
* * * * * * * *
* * * * * * * * *
* * * * * * * * * *
```

Correction

```
#Q1
for i in range(1, 11):
    for j in range(1, i+1):
        print("* ", end ="")
    print()
```

Partie 4 — Boucle While

Ex. 6

L'instruction `while` signifie "tant que". Elle permet d'exécuter un bloc d'instructions, de façon répétée, tant qu'une condition est vraie. Elle sert donc, comme le `for`, à répéter des instructions. Mais, à la différence du `for`, elle permet de répéter ces instructions un nombre indéterminé de fois.

```
while expr :
    # bloc d'instructions
    # exécuté tant que
    # expr est vraie
# suite des instructions
```

Autrement dit: * L'interpréteur évalue la condition. * Si la condition est vraie, il exécute le bloc et retourne évaluer la condition. * Si la condition est fausse il saute le bloc et poursuit la suite du programme.

Bien sûr, un `while` permet de faire ce qu'aurait déjà su faire un `for`, mais comme c'est un peu plus compliqué que le `for` on ne l'utilise généralement pas pour ça:

```
i=10
while  i >= 0:
    print(i,"...")
    i = i - 1
print("Go !!!")
```

Ce qui affiche

```
10 ...
9 ...
8 ...
7 ...
6 ...
5 ...
4 ...
3 ...
2 ...
1 ...
0 ...
Go !!!
```

Le `while` est aussi plus risqué que le `for`, car il faut s'assurer que le programme ne restera pas prisonnier d'une boucle infinie; répétant à jamais le même bloc d'instructions. Il faut donc se convaincre que la condition finira par être fausse.

Voici un exemple simple de boucle infinie:

```
while True:
    print("Hello!")
```

Question 1 : Ecrire un programme qui demande à l'utilisateur "Entrez un nombre entre 1 et 10 :". Si l'utilisateur ne respecte pas la consigne, il affiche "Essayez encore!" et redemande ce nombre. Ainsi de suite jusqu'à ce que l'utilisateur respecte la consigne.

Question 2 : Écrire un programme qui fait la somme des nombres entiers positifs entrés par l'utilisateur, tant que ce dernier souhaite continuer. Il s'arrête dès que l'utilisateur entre un nombre négatif.

Question 2 : Écrire un programme qui demande à l'utilisateur de calculer la suite des i^3 tant que le résultat de ce calcul est inférieur à $1000i^2 + 500i + 7$. En déduire la partie entière de la solution de l'équation $-i^3 + 1000i^2 + 500i + 7 = 0$.

Correction

```
#Q1
n = int(input("Entrez un nombre entre 1 et 10 :"))
while n<0 or n>10:
    print("Essayez encore!")
    n = int(input("Entrez un nombre entre 1 et 10 :"))

#Q2
n = 0
somme = 0
while n >= 0:
    somme += n
    n = int(input("Entrez un nombre positif:"))
print("La somme des nombres positifs est", somme)

#Q3
i=0
while i**3<1000*i**2+500*i+7:
    i += 1
print("La partie entière de la solution est ",i-1)
```

TP3

Partie 1 — Appel de fonction prédéfinies

Ex. 1 — Fonction print

Un des concepts les plus importants en programmation est le concept de fonction. Vous avez déjà manipulé des fonctions dans les précédent TP comme la fonction `print`.

Dans cet exercice, vous allez voir comment appeler une fonction. Une fonction s'appelle en donnant des valeurs que l'on appelle **arguments** entre parenthèses après le nom de la fonction. Si l'appel de fonction a plusieurs arguments, une virgule sépare deux arguments consécutifs.

Un appel de fonction s'écrit donc de la manière suivante :

```
nom_de_la_fonction(argument1, argument2, ...)
```

Les arguments servent à spécifier le comportement de l'appel de la fonction. Par exemple, dans le cas de `print`, les arguments seront les valeurs affichées par la fonction `print`. Les arguments étant des valeurs cela peut correspondre dans le code à :

- des littéraux c'est-à-dire des constantes. Exemples : `15`, `-199`, `12.9`, `"toto"`, `True`, …
- des variables et la valeur ser égale au contenu de la variable. Exemples : `x`, `sum`, …
- un autre appel de fonction. Exemples : `random()`, `sqrt(12)`
- des expressions, c'est-à-dire des valeurs liées par des opérateurs. Exemples : `12 + sum`, `f(12) * 234`, `12 < 24`, `2 * (x+2)`…

Par exemple, pour appeler la fonction `print` pour afficher l'entier `15` on écrit `print(15)`.

La fonction `print` peut être appelée avec un nombre d'arguments quelconque. Les arguments seront affichés dans l'ordre avec un espace les séparant et un saut de ligne à la fin. Un appel `print("La valeur est", value)` affichera donc `La valeur est 100` si `value` est une variable initialisée à `100`.

Vous pouvez remplacer le séparateur par défaut (une chaîne de caractères contenant un espace) par une autre chaîne de caractères (même une chaîne vide), grâce à l'argument `sep`. Il faut pour cela, spécifier la valeur de `sep` en écrivant `sep=valeur` entre les parenthèses de l'appel de fonction.

Exemples :

- `print("Bonjour", "tout", "le", "monde", sep ="*")` affichera `Bonjour*tout*le*monde`

- `print("Bonjour", "tout", "le", "monde", sep ="")` affichera `Bonjourtoutlemonde`

En plus, des arguments qui seront affichés par la focntion `print`, il est possible de changer la chaîne de caractère entre l'affichage de chaque argument (un espace par défaut) ainsi que la chaîne de caractère (un saut de ligne par défaut).

Question 1 : Appeler la fonction `print` pour qu'elle affiche `Hello World!`

Question 2 : Initialiser la variable `your_name` pour qu'elle contienne un chaîne de caractère correpondant à votre nom.

Question 3 : Appeler la fonction `print` pour qu'elle affiche `Hello` suivi d'un espace suivi du contenu de la variable `your_name`.

Question 4 : Utiliser la fonction `print` pour afficher 100 fois le mot `to` sans espace entre les mots.

Correction

```
#Q1
print("Hello World!")

#Q2
your_name = "Arnaud"

#Q3
print("Hello", your_name)

#Q4
for index in range(100):
    print("to", end="")
```

Ex. 2 — Fonction sqrt

En plus des fonctions comme `print` qui effectuent des actions comme afficher du texte, ils existent aussi des fonctions qui calculent des valeurs et **renvoient** (ou **retourne**) donc un résultat.

Un exemple d'une telle fonction est la fonction permettant de calculer la racine carrée d'une valeur. En Python (comme dans beaucoup d'autres langages), cette fonction se nomme `sqrt` qui est une abbriéviation de **square root** qui signifie en anglais la racine carrée.

Un appel de `sqrt(2)` renverra donc la valeur de la racine carrée de 2 qui est environ égal à `1.4142135623730951`.

Pour récupérer la valeur produite par l'appel d'une fonction, il est possible de la stocker dans une variable. Par exemple en écrivant `sqrt2 = sqrt(2)`, on affecte (ou assigne) la valeur (en fait une approximation) de la racine carrée de 2 à la variable `sqrt2`.

On peut aussi utiliser un appel de fonction dans une expression. Par exemple en écrivant `golden_ratio = (1 + sqrt(5))/2`, on affecte la valeur du nombre d'or à la variable `golden_ratio`.

La fonction `sqrt` n'est pas incluse de base dans Python contrairement à la fonction `print`. Elle est dans un module appelé `math` et il faut donc l'importer pour qu'elle soit disponible. La syntaxe (manière d'écrire) pour importer une fonction d'un module est la suivante :

```
from nom_du_module import nom_de_la_fonction
```

Pour importer la fonction `sqrt` du module `math` on a donc ajouter la ligne suivante dans le code :

```
from math import sqrt
```

Il est possible d'importer toutes les fonctions d'un module en mettant `*` en tant que nom de fonction. Par exemple, le code suivant permet d'importer toutes les fonctions du module `math`.

```
from math import *
```

Question 1 : Utiliser la fonction `sqrt` pour initialiser une variable `square_root_of_three` à la valeur de la racine carrée de trois.

Question 2 : Utiliser la fonction `sqrt` pour calculer la distance euclidéenne entre le point de coordonnées (`x1`, `y1`) et le point de coordonnées (`x2`, `y2`) qui représentent deux point dans le plan. Si vous ne connaissez pas la formule permettant de calculer la distance entre deux points, il vous suffit de la rechercher sur internet par exemple au lien suivant.

Correction

```
from math import sqrt

#Q1
square_root_of_three = sqrt(3)

x1 = 0
y1 = 0
```

```
x2 = 1
y2 = 1

#Q2
distance = sqrt((x1 - x2)**2 + (y1 - y2)**2)
```

Partie 2 — Définitions de procédures

Ex. 3 — Définitions de procédures sans paramètres

On vient de voir comment utiliser des fonctions qui existe déjà en Python. Vous allez maintenant définir vos propres fonctions.

La syntaxe pour la définition d'une fonction sans paramètre et donc sans valeur à passer en argument lors d'un appel (on verra dans l'exercice comment rajouter des paramètres) est la suivante :

```
def nom_de_la_fonction():
    ...
    bloc instructions
    ...
```

On peut faire quelques remarques :

- Le mot-clé `def` s'utilise de manière relativement similaire à `if` et `while`. La ligne doit se terminer par un double point suivi d'un bloc d'instructions qui doit être indenté.
- Tout comme pour les noms de variables, vous avez une liberté quasi-totale pour le nom des fonctions. Il vous faut néanmoins respecter les mêmes règles : pas d'espaces (on utilise `_` pour séparer les mots si le nom de la fonction en contient plusieurs), que des minuscules, pas de caractères spéciaux (accents, tilde, parenthèses, accolades, …)

Question 1 : Définir une fonction `print_1000_hello` qui affiche 1000 fois "Hello!" avec un saut à la ligne entre chaque affichage.

Question 2 : Appeler la fonction `print_1000_hello` deux fois.

Question 3 : Définir une fonction `print_1_to_10000` qui affiche les entiers de 1 à 10000 avec un saut à la ligne entre chaque affichage.

Question 4 : Appeler la fonction `print_1_to_10000` trois fois.

Correction

```
#Q1
def print_1000_hello():
    for i in range(1000):
        print("Hello!")


#Q2
for i in range(2):
    print_1000_hello()


#Q3
def print_1_to_10000():
    for i in range(1,10001):
        print(i)


#Q4
for i in range(3):
    print_1_to_10000()
```

Ex. 4 — Définitions de procédures avec paramètres

La syntaxe pour la définition d'une fonction ayant des paramètre est la suivante :

```
def nom_de_la_fonction(nom_parametre1, nom_parametre2, ...):
    ...
    bloc instructions
    ...
```

La liste des paramètres spécifie quelles informations il faudra fournir en guise d'arguments lorsque l'on voudra utiliser cette fonction.

Comme nous l'avons vu à l'exercice précédent les parenthèses peuvent parfaitement rester vides si la fonction ne nécessite pas d'arguments.

Question 1 : Définir une fonction `print_n_hello(n)` qui affiche `n` fois "Hello!" avec un saut à la ligne entre chaque affichage.

Question 2 : Appeler la fonction `print_n_hello(n)` avec comme argument 10.

Question 3 : Écrire une fonction `print_line_multiplication_table(line_number)` qui affiche la ligne d'une table de multiplication, c'est-à-dire tous les multiples de l'entier `line_number` de une fois `line_number` à dix fois `line_number`. Chaque multiple devra être séparé par un caractère `"\t"` qui correspond à une

tabulation. Par exemple, un appel `print_line_multiplication_table(3)` devra afficher la ligne suivante :

```
3   6   9   12  15  18  21  24  27  30
```

Question 4 : Appeler la fonction `print_line_multiplication_table(line_number)` avec comme argument 3.

Question 5 : Écrire une fonction `print_multiplication_table()` qui affiche les lignes 1 à 10 d'une table de multiplication. Un appel `print_multiplication_table()` devra afficher le texte suivant :

```
1   2   3   4   5   6   7   8   9   10
2   4   6   8   10  12  14  16  18  20
3   6   9   12  15  18  21  24  27  30
4   8   12  16  20  24  28  32  36  40
5   10  15  20  25  30  35  40  45  50
6   12  18  24  30  36  42  48  54  60
7   14  21  28  35  42  49  56  63  70
8   16  24  32  40  48  56  64  72  80
9   18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90  100
```

Question 6 : Appeler la fonction `print_multiplication_table()`

Correction

```
#Q1
def print_n_hello(n):
    for i in range(n):
        print("Hello!")

#Q2
print_n_hello(10)

#Q3
def print_line_multiplication_table(line_number):
    for i in range(1, 11):
        print(i * line_number, end="\t")
```

```
    print()

#Q4
print_line_multiplication_table(3)

#Q5
def print_multiplication_table():
    for i in range(1, 11):
        print_line_multiplication_table(i)

#Q6
print_multiplication_table()
```

Partie 3 — Définitions de fonctions

Ex. 5 — Fonctions ayant une unique instruction retour

Comme nous l'avez vu dans la deuxième question de l'exercice 1, ils existent des fonctions comme `sqrt` calculant et renvoyant (ou retournant) un résultat. Afin d'indiquer à l'ordinateur qu'une fonction doit renvoyer une valeur, il faut utiliser le mot-clé `return` suivi de la valeur que l'on souhaite retourner à l'intérieur du code de la fonction.

On rappelle que les valeurs peuvent être exprimées par :

- des littéraux c'est-à-dire des constantes. Exemples : `15`, `-199`, `12.9`, `"toto"`, `True`, …
- des variables et la valeur sera égale au contenu de la variable. Exemples : `x`, `s`, …
- un appel de fonction. Exemples : `random()`, `sqrt(12)`
- des expressions, c'est-à-dire des valeurs liées par des opérateurs. Exemples : `12 + s`, `f(12) * 234`, `12 < 24`, `2 * (x+2)`…

Par exemple le code suivant défini une fonction `product` qui calcule et renvoie le produit de deux nombres `number1` et `number2`.

```
def product(number1, number2):
    return number1 * number2
```

Une fois que la fonction `product` est définie, on peut l'appeler avec des arguments et récupérer la valeur calculée. Considérons le code suivant :

```
p1 = product(12, 10)
P2 = product(p1, 2)
print(p2)
```

1. La variable `p1` prend la valeur du produit de `12` fois `10` soit `120`

2. La variable `p2` prend la valeur du produit de la valeur de `p1` (égale à `120`) et de `2` soit `240`.
3. Le contenu de la variable `p2` est affiché et donc 240 est affiché en sortie.

Question 1 : Définir une fonction `addition` ayant deux paramètres `number1` et `number2` et qui renvoie la somme des deux nombres `number1` et `number2`

Question 2 : Utiliser la fonction `addition` pour calculer la somme de 12 et 14 et mettre le résultat dans la variable `s`.

Question 3 : Afficher le contenu de la variable `s`.

Correction

```
#Q1
def addition(number1, number2):
    return number1 + number2

#Q2
s = addition(12,14)

#Q3
print(s)
```

Ex. 6 — Plusieurs instructions de retour

Dans certains cas, il peut y arriver que le code d'une fonction contienne plusieurs occurences du mot-clé `return`.

Considérons la fonction suivante nommée `find_multiple_in_interval` :

```
def find_multiple_in_interval(value, begin, end):
    for i in range(begin, end):
        if i % value == 0:
            return i
    return -1
```

Cette fonction parcoure tous les entiers compris dans l'interval [begin , end[et si elle trouve un multiple de `value` dans l'intervalle, elle le renvoie. Dés que l'instruction `return` est exécutée, on sort de la fonction. S'il y a plusieurs multiples de `value` dans l'intervalle, seul le premier multiple est renvoyé, car le `return` arrête l'exécution de la fonction et le reste des entiers de l'intervalle n'est pas parcouru. Si à la fin de la boucle, aucun des entiers de l'intervalle étaient un multiple de `value` la valeur `-1` est renvoyé

Question 1 : Écrire la fonction `absolute_value` ayant un paramètre `x` et qui renvoie la valeur absolue de `x`.

Question 2 : Afficher la valeur absolue de `-11`.

Correction

```
#Q1
def absolute_value(x):
    if x < 0:
        return -x
    return x


#Q2
print(absolute_value(-11))
```

Ex. 7 — Variables locales

Lorsque nous définissons des variables à l'intérieur du corps d'une fonction, ces variables ne sont accessibles qu'à la fonction elle-même. On dit que ces variables sont des variables locales à la fonction.

Les variables à l'intérieur de la fonctions sont donc différentes de celles à l'extérieur de la fonction même si elles ont le même nom.

Question 1 : Créer une variable nommée `a` et qui a une valeur égale à `10`.

Question 2 : Définir une fonction `set_a_to_12` sans paramètre et qui initialise une variable `a` avec la valeur `12`.

Question 3 : Appeler la fonction `set_a_to_12`.

Question 4 : Afficher la valeur de la variable `a`.

Correction

```
#Q1
a = 10


#Q2
```

```
def set_a_to_12():
    a = 12


#Q3
set_a_to_12()


#Q4
print(a)
```

Ex. 8 — Appel par nom

Lorsqu'une variable est passé comme argument d'un appel de fonction, c'est seulement sa valeur qui est transmise et pas la variable elle-même. Un appel de fonction ne peut donc pas modifié la valeur d'une variable. Vous allez pouvoir vérifier cette affirmation avec cet exercice.

Attention vous verrez par la suite des objets (les liste par exemple) dont l'état sera modifiable y compris par un appel de fonction.

Question 1 : Créer une variable `a` ayant pour valeur `0`.

Question 2 : Définir un fonction `increment` ayant un paramètre `value`. Cette fonction devra augmenter de un la valeur de `value` et renvoyer la nouvelle valeur de `value`.

Question 3 : Appeler la fonction `increment` avec comme argument `a`.

Question 4 : Afficher la valeur de `a`.

Question 5 : Appeler la fonction `increment` avec comme argument `a` et stocker le retour de la focntion dans la variable `a`.

Question 6 : Afficher la valeur de `a`.

Correction

```
#Q1
a = 0


#Q2
def increment(value):
```

```
    value += 1
    return value


#Q3
increment(a)


#Q4
print(a)


#Q5
a = increment(a)


#Q6
print(a)
```

Partie 5 — Complex functions

Ex. 9 — Géométrie

Question 1 : Écrire une fonction `rectangle_area` ayant deux paramètres `height` et `length` et qui renvoie l'aire du rectangle correspondant.

Question 2 : Écrire une fonction `rectangle_perimeter` ayant deux paramètres `height` et `length` et qui renvoie le périmètre du rectangle correspondant.

Question 3 : Écrire une fonction `triangle_area` ayant trois paramètres `side1`, `side2` et `side3` (les longueurs des trois cotés) et qui renvoie l'aire du triangle correspondant. On utilisera pour cela la formule de Héron.

Question 4 : Écrire une fonction `triangle_perimeter` ayant trois paramètres `side1`, `side2` et `side3` (les longueurs des trois cotés) et qui renvoie le périmètre du triangle correspondant.

Question 5 : Écrire une fonction `disk_area` ayant un paramètre `radius` et qui renvoie l'aire du disque correspondant. Pour la valeur de pi, on utilisera `pi` qu'on a importé du module math (deuxième ligne du code).

Question 6 : Écrire une fonction `disk_perimeter` ayant un paramètre `radius` et qui renvoie le périmètre du disque correspondant.

Correction

```
from math import sqrt
from math import pi


#Q1
def rectangle_area(length, width):
    return length * width

#Q2
def rectangle_perimeter(length, width):
    return 2 * (length + width)

#Q3
def triangle_area(side1, side2, side3):
    p = (side1 + side2 + side3)/2
    return sqrt(p * (p-side1) * (p-side2) * (p-side3))

#Q4
def triangle_perimeter(side1, side2, side3):
    return side1 + side2 + side3

#Q5
def disk_area(radius):
    return pi * (radius ** 2)

#Q6
def disk_perimeter(radius):
    return 2*pi*radius
```

Ex. 10 — Suites

Question 1 : Écrire une fonction `sum_from_begin_to_end` ayant deux paramètres `begin` et `end` et qui renvoie la somme des entiers de `begin` à `end-1` compris.

Question 2 : Écrire une fonction `fibonacci` ayant un paramètre `n` et qui renvoie le terme de rang `n` de la suite de Fibonacci.

Question 3 : Écrire une fonction `syracuse_next_term(term)` qui calcule le terme suivant de la suite de Syracuse à partir de la valeur `term` du terme de la suite passée en paramètre. utiliser la division entière (opérateur `//`en Python) afin de n'avoir que des entiers.

Question 4 : Écrire une fonction `print_syracuse_values(initial_term)` qui affiche tous les termes de la suite commençant par le terme initial `initial_term` jusqu'au premier terme égal à 1 compris.

Question 5 : Appeler la fonction `print_syracuse_values` avec comme argument la valeur `15`.

Correction

```
#Q1
def sum_from_begin_to_end(begin, end):
    s = 0
    for i in range(begin, end):
        s += i
    return s

#Q2
def fibonacci(n):
    term = 0
    next_term = 1
    for i in range(n):
        temp = next_term
        next_term += term
        term = temp
    return term

#Q3
def syracuse_next_term(term):
    if term % 2 == 0:
        return term//2
    return 3*term + 1

#Q4
def print_syracuse_values(initial_term):
    term = initial_term
    while term != 1:
        print(term)
        term = syracuse_next_term(term)
    print(term)

#Q5
print_syracuse_values(15)
```

TP4

Partie 1 — Théorie des nombres

Ex. 1 — Primalité d'un entier

Rappels: - Un entier `n` est premier si il n'est divisible par aucun entier `i` tel que `1<i<n`. - Pour savoir si un nombre est premier, il suffit de tester la divisibilité de `n` par tous les entiers entre `2` et `n/2` (arrondi à l'inférieur). - Si aucun diviseur de `n` n'est trouvé parmi les entiers candidats, cela signifie que `n` est premier. - Dès lors qu'un diviseur `n` est trouvé parmi les candidats, alors le nombre n'est pas premier.

Par exemple: - `11` n'est divisible par aucun des nombres `2`, `3`, `4`, et `5`, donc il est premier. - `18` est divisible par `3`, donc il n'est pas premier.

Quelques outils utiles: - `a` est divisible par `b` lorsque le reste de la division entière de `a` par `b` vaut `0`. En python, utiliser l'opérateur `%`. - Pour effectuer une division entière, autrement dit la division arrondie à l'entier inférieur, il faut utiliser l'opérateur `//`. - Pour énumérer l'ensemble des candidats possibles de diviseurs, vous utiliserez la fonction `range(x,y)` qui énumère tous les nombres de `[x,y[`. Attention au fait que `y` n'est pas énuméré. Si vous en avez besoin, appelez range(x,y+1).

Question 1 : Écrire une fonction `is_prime` ayant un paramètre `n`, qui renvoie `True` lorsque `n` est premier et `False` sinon.

Question 2 : Écrire une fonction `print_prime` ayant un paramètre `n`, faisant appel à la fonction `is_prime` et qui affiche soit: - une chaîne de la forme "Le nombre 5 est premier." pour les nombres premiers. - une chaîne de la forme "Le nombre 6 n'est pas premier." sinon.

Question 3 : Appeler la fonction `print_prime` avec comme argument la valeur `4`.

Question 4 : Appeler la fonction `print_prime` avec comme argument la valeur `7`.

Question 5 : Appeler la fonction `print_prime` avec comme argument la valeur `15`.

Question 6 : Appeler la fonction `print_prime` avec comme argument la valeur `101`.

Question 7 : Écrire une fonction `print_all_prime` ayant un paramètre `n`, qui affiche la suite de tous les entiers premiers entre `2` et `n`. Par exemple, `print_all_prime(11)` affichera: `2 3 5 7 11`

Question 8 : Appeler `print_all_prime` avec comme argument la valeur `101`.

Correction

```
# Q1
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, n//2+1):
        if n % i == 0:
            return False
    return True


# Q2
def print_prime(n):
    if is_prime(n):
        print("Le nombre", n, "est premier.")
    else:
        print("Le nombre", n, "n'est pas premier.")


# Q3
print_prime(4)


# Q4
print_prime(7)


# Q5
print_prime(15)


# Q6
print_prime(101)


# Q7
def print_all_prime(n):
    for i in range(2, n+1):
        if is_prime(i):
            print(i)
```

```
# Q8
print_all_prime(101)
```

Ex. 2 : Décomposition en facteurs premiers

Le Problème: - La décomposition en facteurs premiers d'un entier `n` est la liste des nombres premiers dont le produit vaut `n` (par exemple: `20=2*2*5`) - On veut ici écrire une fonction 'decompose' affichant les différents facteurs le composant, dans l'ordre croissant. - Pour cela, on va avoir besoin d'une variable `i`, qui au départ vaudra 2. on va tester si `i` divise `n`. Tant que l'on n'a pas trouvé tous les diviseurs (`n` n'est pas égal à `1`) - Si `i` divise `n`, cela signifie que `i` est un diviseur de `n`. Il faut alors: - Afficher `i` - Diviser `n` par `i` - Ne pas modifier `i` (sinon on va oublier de vérifier si `i` n'apparaitrait pas plusieurs fois dans la décomposition) - Si `i` ne divise pas `n`, on ajoute `1` à la valeur de `i`

Question 1 : Écrire une fonction `decompose` ayant un paramètre `n`, qui affiche une première ligne du type "L'entier 12 se décompose en:", suivi de la liste des facteurs premiers de `n`, un par ligne et du plus petit au plus grand.
Ainsi, `decompose(12)` doit afficher sur quatre lignes:

```
L'entier 12 se décompose en:

2
2
3
```

Question 2 : Appeler la fonction `decompose` avec comme argument la valeur `6`.

Question 3 : Appeler la fonction `decompose` avec comme argument la valeur `19`.

Question 4 : Appeler la fonction `decompose` avec comme argument la valeur `90`.

Correction

```
# Q1
def decompose(n):
    print("L'entier", n, "se décompose en:")
```

```
    i = 2
    while n != 1:
        if n % i == 0:
            print(i)
            n = n // i
        else:
            i = i + 1


# Q2
decompose(6)


# Q3
decompose(19)


# Q4
decompose(90)
```

Ex. 3 Calcul du PGCD par l'Algorithme d'Euclide

- Le pgcd de deux entiers `a` et `b` est le plus grand entier `n` qui divise `a` et `b`.
- L'algorithme d'Euclide permet de le calculer, selon le principe suivant:
  - Si `a` est divisible par `b`, le pgcd de `a` et `b` vaut `b`.
  - Sinon, soit `r` le reste de la division entière de `a` par `b`
  - Le pgcd de `a` et `b` est égal au pgcd de `b` et `r`

Par exemple, le calcul du PGCD de `90` et de `35` passe par les étapes suivantes:

```
a  = b  x q + r
90 = 35 x 2 + 20
35 = 20 x 1 + 15
20 = 15 x 1 + 5
15 = 5  x 3 + 0
```

Le PGCD de `90` et `35` est donc `5`.

Question 1 : Écrire une fonction `euclide` ayant deux paramètres `a` et `b`, qui affiche les différentes étapes du calcul du pgcd de `a` et de `b`.

```
L'affichage sera composé d'une première ligne donnant les arguments de la fonction, de la su

Par exemple, `euclide(90,35)` affichera:

```
```
```

```
Calcul du PGCD de 90 et de 35 :
90 = 35 x 2 + 20
35 = 20 x 1 + 15
20 = 15 x 1 + 5
15 = 5 x 3 + 0
Le PGCD est donc 5
```

Question 2 : Appeler la fonction `euclide` avec comme arguments `18` et `12`.

Question 3 : Appeler la fonction `euclide` avec comme arguments `7` et `5`.

Correction

```
# Q1
def euclide(a, b):
    print("Calcul du PGCD de", a, "et de", b, ":")
    r = 1  # initialiser r à une valeur différente de 0 pour rentrer au moins une fois dans
    while r != 0:
        r = a % b
        q = a // b
        print(a, "=", b, "x", q, "+", r)

        a = b
        b = r

    pgcd = a
    print("Le PGCD est donc", pgcd)


# Q2
euclide(18, 12)


# Q3
euclide(7, 5)
```

Partie 2 — Arithmétique

Ex. 4 — Exponentiation Rapide

Le but de cet exercie est d'écrire une fonction réalisant l'opérateur d'exponentiation (x**n) en n'utilisant que les opérations + et *

- L'exponentiation d'un entier `x` par un entier `n` est l'entier xn=`x*x*...*x` (`n` fois).
- L'algorithme d'exponentiation rapide permet de calculer xn efficacement, selon le principe suivant:
  - Si `n` est égal à `0`, xn = 1
  - Si `n` est égal à `1`, xn = x
  - Si `n` est pair, alors `n` peut s'écrire `2*m` (division entière), et xn = xm * xm
  - Si `n` est impair, alors `n` peut s'écrire `2*m+1`, et xn = xm * xm * x

On utilisera une variable `resultat` initialisée à `1`. Si `n` est pair, on peut faire x=x2 et n=n/2 sans changer le résultat. Si `n` est impair, on peut multiplier `resultat` par `x`, puis faire x=x2 et n=(n-1)/2.

Question 1 : Écrire une fonction `exponentiation_rapide` ayant deux paramètres `x` et `n`, qui retourne l'entier xn, calculé par exponentiation rapide.

Question 2 : Appeler et afficher le résultat de la fonction `exponentitiation_rapide` avec comme arguments `3` et `4`.

Question 3 : Appeler et afficher le résultat de la fonction `exponentitiation_rapide` avec comme arguments `2` et `10`.

Correction

```
# Q1
def exponentiation_rapide(x, n):
    if n == 0:
        return 1
    resultat = 1
    while n > 1:
        if n % 2 == 0:
            x = x * x
            n = n // 2
        else:
            resultat = resultat * x
            x = x * x
            n = (n-1) // 2
    return resultat * x
```

```
# Q2
print(exponentiation_rapide(3, 4))


# Q3
print(exponentiation_rapide(2, 10))
```

Ex. 5 — Suite de Syracuse

- La suite de Syracuse depuis `x` est la suite (un) telle que:
  - u0 = x
  - Si `n` est pair, alors un+1 = un/2
  - Si `n` est impair, alors un+1 = 3*un+1
- On observe que quel que soit `x`, après un certain nombres d'étapes `n`, un=`1`, Après ce `n`, la suite alterne entre trois valeurs: `4`, `2` et `1`.
- L'objectif ici est d'afficher les termes de la suite depuis `x`, jusqu'à la première apparition de la valeur `1`.

Question 1 : Écrire une fonction `syracuse` ayant un paramètre `x`, qui affiche les entier produits par la suite de syracuse initialisée en `x`, jusqu'à atteindre `1`. Ainsi, `syracuse(10)` doi afficher sur sept lignes:

```
10
5
16
8
4
2
1
```

Question 2 : Appeler la fonction `syracuse` avec comme argument la valeur `4`.

Question 3 : Appeler la fonction `syracuse` avec comme argument la valeur `13`.

Question 4 : Appeler la fonction `syracuse` avec comme argument la valeur `25`.

Correction

```
# Q1
def syracuse(n):
    while n != 1:
```

```
        print(n)
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
    print(n)


# Q2
print("Question 2:")
syracuse(4)


# Q3
print("Question 3:")
syracuse(13)


# Q4
print("Question 4:")
syracuse(25)
```

TP5

Partie 1 — Les Tuples

Ex. 1 — Les Tuples

En python, on peut créer des tuples (aussi appelées vecteurs), par exemple le tuple `(12,7)` est une paire, qui contiens 2 valeurs: les entiers `12` et `7`.

Question 1 (console) : Ouvrir la console python en bas à gauche, et y entrer les commandes suivantes:

```
>>> t = (12, 7, 26)
>>> print(t)
>>> type(t)
>>> t[0]
>>> t[1]
>>> t[2]
>>> t[10]
>>> t[3]
>>> print(t)
>>> len(t)
```

Explications: - Il existe un type `tuple`, composé de plusieurs 'cases', chacune contenant une valeur - On peut accéder à ces valeurs avec les crochets (`t[0]`, `t[1]` …) - `t[i]` renvoie la valeur contenue dans la i-ème case du tuple `t`, mais attention, la numérotation des cases commence à `0` - Si on essaye d'accéder à un numéro de case trop grand, une Erreur est renvoyée - Pour un tuple de `n` éléments, les cases sont donc numérotées de `0` à `n-1`, et on ne peut donc pas accéder à `t[n]` - `len` est une fonction qui prends un tuple en argument et renvoie sa taille (length en anglais)

Question 2 (console) : On continue dans la console:

```
- t = (12, 2.1, "toto")
- print(l[0])
- print(l[1])
- print(l[2])
- print(l)
- t1 = (1, 2)
- t2 = (3, 4)
- print(t1 + t2)
```

Explications: - Les tuples peuvent contenir plusieurs éléments de type différents, içi un `int` (entier), un `float` (nombre réel) et un `str` (chaîne de caractères) - L'addition de deux tuples renvoie un tuple les fusionnant

Question 3 : Créer un tuple `my_tuple` à deux éléments, `0` et `100`

Question 4 : Afficher son deuxième élément en utilisant la fonction `print`

Question 5 : Afficher sa taille avec `len`

Correction

```
# Q1 et Q2 se font dans la console

# Q3

my_tuple = (0, 100)

# Q4

print(my_tuple[1])

# Q5

print(len(my_tuple))
```

Ex. 2 — Fonctions

Question 1 : On souhaite écrire une fonction notes_extremes, qui demande 5 notes (entre 0 et 20) à l'utilisateur, et en extrait la note la plus basse et la note la plus haute. - Le mot-cef `return`, vu précédement, permet à une fonction de renvoyer un résultat, on souhaite içi l'utiliser pour en renvoyer deux - on va donc utiliser un tuple à deux éléments!

Question 2 : Observer l'exemple fourni. Le comportement est similaire, à celui de la question 1, mais dans ces cas précis on peut ommetre les parenthèses formant le tuple: - `return a, b, c, ...` retourne le tuple `(a, b, c, ...)` - `x, y, ... = <tuple>` permet d'assigner directement aux varibles `x`, `y` … les valeurs de `<tuple>[0]`, `<tuple>[1]` …

Correction

```
# Exercice 2

# Q1


def note_extremes():
    note_min = 20
    note_max = 0
    for i in range(1, 6):
        note = int(input("Note "+str(i)+"?"))
        if note < note_min:
            note_min = note
        if note > note_max:
            note_max = note
    paire_resultat = (note_min, note_max)
    return paire_resultat


pair_notes = note_extremes()

a = pair_notes[0]
b = pair_notes[1]

print("La note la plus basse est", a, "et la note la plus haute est", b)

# Q2
```

```
def exemple():
    return 0, "bonjour"


a2, b2 = exemple()

print(a2, b2)
```

Partie 2 — Les Listes

Ex. 3 — Les Listes

En python, on peut créer des listes (aussi appelées tableaux), par exemple la liste `[12,7,26]` contiens 3 valeurs: les entiers `12`, `7` et `26`.

Question 1 (console) : Ouvrir la console python en bas à gauche, et y entrer les commandes suivantes:

```
>>> l = [12, 7, 26]
>>> print(l)
>>> type(l)
>>> l[0]
>>> l[1]
>>> print(l)
>>> len(l)
```

Explications : - Il existe un type `list`, composé de plusieurs 'cases', chacune contenant une valeur - Comme pour les tuples, `l[i]` renvoie la valeur contenue dans la i-ème case de la liste `l`, et la numérotation des cases commence à `0` - On parle d'indice et pas de numéro de case, ainsi, la première case est d'indice `0`, la deuxième d'indice `1`, la dernière d'indice `n-1` - La fonction `len` fonctionne aussi sur les listes

Question 2 (console) : On continue dans la console:

```
>>> l = [12, 7, 26]
>>> print(l[1])
>>> l[1] = 77
>>> print(l)
```

Explications: - On peut modifier les valeurs contenues dans les cases d'une liste, en utilisant l'assignation (le `=`) - `l[a] = b` replace ainsi la valeur dans la case d'indice `a` par `b` - Cela ne fonctionne pas avec les tuples : les tuples ne sont pas modifiables une fois créés

Question 3 (console) : On continue dans la console:

```
>>> l = [12]
>>> l.append(1)
>>> print(l)
>>> l.append(42)
>>> print(l)
>>> print("taille de l:",len(l))
>>> l.pop()
>>> print(l)
>>> elem = l.pop()
>>> print(elem)
>>> print(l)
```

Explications : - La taille d'une liste peut être modifiée, ainsi, on peut lui ajouter et lui enlever des cases - append permet d'ajouter un élément en bout de liste - pour l'utiliser on fait nom_de_la_liste.append(element_a_inserer) - append n'est pas une fonction comme celles que vous avez vu jusqu'à présent, elle est appelée depuis une variable liste (c'est ce que le variable.append signifie) et agit sur celle-ci, on dit que append est une méthode des listes - Il existe d'autres méthodes, comme 'pop' - faire nom_de_la_liste.pop() à deux effet: il supprime la dernière case de la liste, et return la valeur contenue dans la case supprimée

Question 4 (console) : On continue dans la console:

```
>>> l = []
>>> print(l)
>>> l = [10, 20, 30, 40]
>>> l.insert(0,1)
>>> print(l)
>>> l.insert(2,25)
>>> print(l)
>>> l.pop(2)
>>> print(l)
>>> del l[0]
>>> print(l)
```

Explications : - On peut créer une liste vide, qui ne contiens aucune case avec `[]` - On peut insérer un élément en milieu de liste avec .insert, qui prends en arguments le numéro de case où insérer (son indice), et la valeur à insérer - Si 'l' est une variable liste de taille `n`, insérer une valeur `v` dans la case d'indice `0` (en faisant `l.insert(0,v)`) ajoute la nouvelle case devant la liste - Insérer en indice `1` insère entre les cases d'indice `0` et `1`, et ainsi de suite - Insérer 'v' en case d'indice `n` a le même effet que faire `l.append(v)` - On a vu précédemment que `l.pop()` supprime la dernière case de la liste et en renvoie la valeur - On peut aussi supprimer des case en milieu de liste, en précisant quel indice de case supprimer (`l.pop(2)` supprime la case d'indice `2`, donc la troisième case puisqu'on commence à `0`) - `del l[i]` permet aussi de supprimer la case d'indice `i`. Contrairement à `.pop`, l'élément supprimé n'est pas renvoyé.

Question 5 (console) On continue dans la console:

```
>>> l1 = [1, 2]
>>> l2 = [3, 4, 5]
>>> l3 = l1 + l2
>>> print(l1,l2,l3)
>>> l1.extend(l2)
>>> print(l1,l2)
```

Explications : - Il y a plusieurs manières de fusionner des listes - Utiliser l'addition permet de le faire en créant une nouvelle liste, sans modifier les listes que l'ont fusionne - La méthode `.extend` permet d'étendre une liste en y ajoutant une autre, cela reviens à itérer le `.append`

Question 6 : Créer une variable `ma_liste` contenant les quatre valeurs `11`, `7`, `2` et `5`

Question 7 : En utilisant `print`, afficher `ma_liste`

Question 8 : Afficher la valeur contenue dans la première case de `ma_liste`

Question 9 : Afficher la valeur contenue dans la deuxième case de `ma_liste` (d'indice `1`)

Question 10 : Afficher la valeur d'indice `3` dans `ma_liste`

Question 11 : Mettre la valeur 1 dans la première case

Question 12 : Ajouter une case en bout de liste, contenant la valeur `57`

Question 13 : Ajouter une case en début de liste, contenant la valeur `-1`

Question 14 : Supprimer la troisième case de la liste

Question 15 : Afficher `ma_liste`

Question 16 : Afficher le nombre de cases dans `ma_liste` (sa taille)

Correction

```
# Exercice 1

# Q1

ma_liste = [11, 7, 2, 5]

# Q6

print(ma_liste)
```

```
# Q7

print(ma_liste[0])

# Q8

print(ma_liste[1])

# Q9

print(ma_liste[3])

# Q10

ma_liste[0] = 1

# Q11

ma_liste.append(57)

# Q12

ma_liste.insert(0, -1)

# Q13

del ma_liste[2]

# Q14

print(ma_liste)

# Q15

print(len(ma_liste))
```

Ex. 4 — Parcours

On a vu, lors des TP précédents, que faire :

```
for i in range(10):
    <code>
```

Exécute `<code>` 10 fois, pour i allant de 0 à 9.

Si `l` est une liste de taille 10, ses cases sont indexées de 0 à 9.

On peut donc itérer sur les indices des cases d'une liste `l`, et le code:

```
for i in range(len(l)):
  print(l[i])
```

affiche le contenu de `l`, une valeur par ligne.

On dit qu'on utilise une boucle for pour parcourir la liste `l`

On peut aussi itérer directement sur les valeurs contenues dans les cases d'une liste `l`:

```
for v in l:
  print(v)
```

Essayez d'utiliser ces deux parcours en console.

Question 1 : Créer une variable `ma_liste` contenant une liste vide

Question 2 : Y ajouter les entiers de 1 à 30, avec une boucle `for` et `.append`

Question 3 : Compter combien d'entiers pairs sont dans `ma_liste`, et mettre le résultat dans une variable `nb_pairs`

Question 4 : Remplacer chaque entier pair dans `ma_liste` par la valeur 1

Question 5 : Afficher `ma_liste`

Correction

```
# Q1

ma_liste = []

# Q2

for i in range(1, 31):
    ma_liste.append(i)

# Q3

nb_pair = 0

for v in ma_liste:
    if v % 2 == 0:
```

```
        nb_pair += 1

print("La liste", ma_liste, "contiens", nb_pair, "valeurs paires")

# Q4

for i in range(len(ma_liste)):
    if ma_liste[i] % 2 == 0:
        ma_liste[i] = 1

# Q5

print(ma_liste)
```

Ex. 5 — Moyenne

Les listes peuvent être utilisées comme argument de fonction:

```
def fonction(liste):
    <code>
```

Question 1 : Écrire une fonction `moyenne` qui prends en argument une liste (non vide) de nombres, et en `return` la moyenne.

Question 2 : Appeler `moyenne` sur la liste `[12,18,2.5,10]` et en afficher le résultat.

Correction

```
# Q1

def moyenne(l):
    total = 0
    for v in l:
        total += v
    return total / len(l)


# Q2

print(moyenne([12, 18, 2.5, 10]))
```

Ex. 6 — Sous-listes

Question 1 (console) : On continue dans la console:

```
>>> l = [12, 7, 26, 2, 56]
>>> print(l[0])
>>> print(l[0:2])
>>> print(l[1:4])
>>> print(l[1:2])
>>> print(l[2:2])
```

Explications : - A partir d'une liste `l`, on peut extraire une sous-liste, en utilisant `l[i:j]` - `l[i:j]` contiens les éléments de `l`, entre les indices `i` et `j-1` - Ainsi, `l[i:i+1]` est équivalent à `l[i]`, et `l[i:i]` est vide - La sous-liste extraite est indépendante

Question 2 (console) : On continue dans la console:

```
>>> l = list(range(22,27))
>>> print(l[:3])
>>> print(l[3:])
```

Explications : - `list(range(...))` permet de créer facilement une liste d'entiers consécutifs - `l[:j]` contiens les éléments de `l`, entre les indices `0` et `j-1` - `l[i:]` contiens les éléments de `l`, entre les indices `i` et la fin de la liste

Question 3 (console) : On continue dans la console:

```
>>> x = list(range(5))
>>> y = x
>>> print(x,y)
>>> y.append(5)
>>> print(x,y)
>>> y = x.copy()
>>> print(x,y)
>>> y.append(5)
>>> print(x,y)
>>> y = x[:2]
>>> y[0] = 10
>>> print(x, y)
```

Explications : - Lorsque l'on fait `y = x`, la variable `y` designe la même liste que la variable `x`. - Si l'une des deux est modifiée, les deux se retrouvent changées. - Si on souhaite éviter ce comportement, on utilise `.copy()` qui met en `y` une copie (indépendante) de la liste qui se trouve dans `x` - Modifier une sous-liste n'affecte pas sa liste parente, une sous-liste est donc une copie (partielle).

Question 4 : Créer une liste `x` contenant les entiers de `0` à `9`

Question 5 :En extraire une sous-liste `y` contenant sa deuxième moitié (les entiers de `5` à `9`) en utilisant les sous-listes

Question 6 : Afficher `x` et `y`

Correction

```
# Q1, Q2, Q3 se font dans la console.

# Q4

x = list(range(10))

# Q5

y = x[5:10]

# Q6

print(x, y)
```

Ex. 7 — Copies de listes

Question 1 : Ecrire une fonction `ma_fonction`, qui prend en argument une liste d'entiers positifs, et a le comportement suivant: - On trouve les indices correspondants a l'élément maximal de la liste (la plus grande valeur, qui peut apparaitre plusieurs fois) - On remplace par `0` le contenu de toutes les cases correspondantes - On affiche la liste obtenue sous la forme: `resultat <liste>`

Question 2 : Suivre les instructions suivantes

```
- Créer une liste `ma_liste`, contenant `[1, 4, 56, 3, 45, 56, -2, 7]`
- Aficher `ma_liste`
- Appeler `ma_fonction` avec comme argument `ma_liste`
- Aficher `ma_liste`
- réaffecter `ma_liste` à `[1, 3, 2]`
- Appeler de nouveau `ma_fonction`, mais sur `ma_liste.copy()` cette fois-ci
- Aficher `ma_liste`
```

Remarque: - On constate que si une liste est modifiée dans une fonction, elle est également modifiée en dehors de la fonction. - La méthode `.copy()` permet d'éviter ce comportement (si l'on souhaite que la liste passée en argument ne soit pas modifiée)

Correction

```
# Exercice 5


# Q1

def ma_fonction(l):
    valeur_max = 0
    for v in l:
        if v > valeur_max:
            valeur_max = v
    for i in range(len(l)):
        if l[i] == valeur_max:
            l[i] = 0
    print("resultat", l)


# Q2

ma_liste = [1, 4, 56, 3, 45, 56, -2, 7]

print(ma_liste)

ma_fonction(ma_liste)

print(ma_liste)

ma_liste = [1, 3, 2]

ma_fonction(ma_liste.copy())

print(ma_liste)
```

Partie 3 — Les Dictionnaires

Ex. 8 — Les Dictionnaires

En python, on peut aussi créer des dictionnaires (aussi appelées tableaux associatif).

Question 1 (console) : Ouvrir la console python en bas à gauche, et y entrer les commandes suivantes:

```
>>> dico = {"Alice":1.75, "Bob":1.8, "Charlie":1.72}
>>> print(dico)
>>> type(dico)
>>> dico["Alice"]
>>> dico["Bob"]
>>> dico["David"]
>>> len(dico)
```

Explications : - Il existe un type `dict`, qui associe des eléments à d'autres. On parle de clefs et de valeurs. - Par exemple, le dicitionnaire `dico` associe à la clef `"Alice"` la valeur `1.72` - On peut accéder à la valeur d'une clef avec les crochets `dico[...]` - Si on essaye d'accéder à une clef inconnue, une Erreur est renvoyée - `len` fonctionne aussi sur les dictionnaires, et compte le nombre de paires (clef:valeur) contenues

Question 2 (console) : On continue dans la console:

```
- dico = {}
- dico["Alice"] = "Boulanger"
- dico["Bob"] = "Instituteur"
- dico["Charlie"] = "Charpentier"
- print(dico)
- dico["Alice"] = "Comptable"
- del dico["Charlie"]
- dico.pop("Bob")
- print(dico)
```

Explications : - On peut ajouter des clefs et des valeurs en utilisant `dico[clef] = valeur` - Si `clef` est inconnue, une case l'associant à `valeur` est ajoutée au dictionnaire - Si `clef` est déjà une clef de `dico`, l'ancienne valeur est remplacée par la nouvelle - Comme pour les, liste, on peut supprimer des éléments avec `del` - Les dictionnaires possèdent une méthode `.pop(clef)` permettant d'en supprimer une clef et sa valeur. Comme pour les listes, la valeur associée à l'élément supprimé est renvoyé en résultat.

Question 3 : Créer un dictionnaire `my_dict` initialement vide.

Question 4 : Associer à la clef `"Alice"` la valeur `"06 23 45 67 89"`

Question 5 : Associer à la clef `"Bob"` la valeur `"06 98 76 54 32"`

Question 6 : Changer la valeur d'`"Alice"` à `"07 22 44 66 88"`

Question 7 : Afficher `my_dict`

Question 8 : Afficher sa taille

Correction

```
# Q3

my_dict = {}

# Q4

my_dict["Alice"] = "06 23 45 67 89"

# Q5

my_dict["Bob"] = "06 98 76 54 32"

# Q6

my_dict["Alice"] = "07 22 44 66 88"

# Q7

print(my_dict)

# Q8

print(len(my_dict))
```

Ex.9 — Parcours

Il y à trois manières d'itérer sur les éléments d'un dictionnaire avec une boulce `for`.

- On peut itérer sur les clefs :

  ```
  for clef in dico.keys():
    print(clef)
  ```

- On peut itérer sur les valeurs associées au clefs :

  ```
  for valeur in dico.values():
    print(valeur)
  ```

- Et enfin on peut itérer sur les deux en même temps :

  ```
  for clef, valeur in dico.items():
    print(clef,":",valeur)
  ```

Essayez d'utiliser ces parcours en console.

Question 1 : Créer une variable `parite` contenant un dictionnaire vide.

On souhaite y stocker des paires clef:valeur sous la forme suivante: Les clefs seront des entiers, les valeurs seront des chaînes de caractères en indiquant la parité (`"pair"` ou `"impair"`).

Question 2 : Y ajouter les clefs de 10 à 20, avec leur valeurs associées.

Question 3 : Afficher le contenu de `parite`, en affichant en console une ligne `"L'entier <i> est <pair/impair>"` par entrée dans le dictionnaire.

Correction

```
# Q1

parite = {}

# Q2

for i in range(10, 21):
    if i % 2 == 0:
        parite[i] = "pair"
    else:
        parite[i] = "impair"


# Q3

for c, v in parite.items():
    print("L'entier", c, "est", v)
```

TP7

Partie 1 — Recherche

Ex. 1

Recherche dans un tableau non trié

On veut chercher une valeur dans un tableau (également appelé une liste en Python) qui contient une suite d'entiers non triés.

Dans ce cas, il n'y a pas d'algorithme plus efficace que de parcourir le tableau jusqu'à trouver la valeur cherchée, puisqu'elle peut se trouver dans n'importe quelle position. Cet algorithme a une complexité O(n), ou n est la longeur du tableau, puisque dans le pire des cas il faut en parcourir la totalité pour vérifier que la valeur cherchée n'y apparait pas.

On appelle ce type de recherche une « recherche linéaire ».

Question 1 : Écrire le code de la fonction `linear_search(x, a)`. La fonction doit retourner un entier (entre `0` et `len(a) - 1`) qui correspond à la position de l'élément `x` dans le tableau `a` (ou de sa première occurrence, si `x` apparait plusieurs fois dans `a`), ou bien l'entier `-1` si `x` n'apparait pas dans `a`. Utiliser l'algorithme de recherche linéaire pour cet exercice.

Question 2 : Pour tester la fonction `linear_search`, afficher les résultats de la recherche des valeurs `2`, `5`, `10` et `6` dans le tableau `a1`.

Recherche dans un tableau trié

Si le tableau où on veut chercher la valeur est déjà trié, on peut utiliser un algorithme plus efficace que la recherche linéaire, c'est-à-dire la recherche dichotomique.

On commence par chercher la valeur dans tout le tableau, donc de la position `0` à la position `n - 1`. On calcule la position moyenne `m` entre ces deux positions et on vérifie si la valeur qu'on cherche est dans la position `m`. Si c'est le cas, on termine en retournant `m` comme résultat.

Si ce n'est pas le cas, on continue de chercher, soit dans la moitié du tableau qui reste à la gauche de la position `m` (si la valeur qu'on cherche est inférieure à l'élément du tableau en position `m`), soit dans la moitié droite (si elle est supérieure).

On continue de chercher jusqu'à trouver la valeur au milieu d'un sous-tableau ; si on arrive à éliminer tous les éléments du tableau sans l'avoir trouvée, on retournera `-1` pour indiquer que la valeur n'apparaît pas dans le tableau.

Comme on a vu en cours d'Introduction à l'informatique, cet algorithme élimine à chaque étape à peu près la moitié du tableau, ce qui signifie qu'il termine en log n étapes ou, en symboles, en temps O(log n).

Question 3 : Écrire le code de la fonction `binary_search`. Comme pour `linear_search`, cette fonction doit retourner un entier (entre `0` et `len(a) - 1`) qui correspond à la position de l'élément `x` dans le tableau `a`, ou bien l'entier `-1` si `x` n'apparaît pas dans `a`. Utiliser l'algorithme de recherche dichotomique décrit ci-dessus.

Question 4 : Pour tester la fonction `linear_search`, afficher les résultats de la recherche des valeurs `2`, `5`, `10` et `6` dans le tableau trié `a2`.

Comparaison des temps de calcul

Pour mesurer le temps de calcul d'un morceau de code en Python on peut utiliser la fonction `default_timer()`, qui retourne un `float` representant le temps actuel en secondes :

```
start = default_timer()
# insérer le code ici
end = default_timer()
```

On obtiendra (une approximation du) temps d'execution du code en calculant `end - start`, le difference entre le temps initial et final.

Pour comparer le temps de calcul des fonctions `linear_search` et `binary_search` et apprecier la différence entre un algorithme à temps linéaire et un algorithme à temps logarithmique, on va mesurer combien de seconds prennent les deux fonctions dans le pire des cas (c'est-à-dire, quand la valeur cherchée n'apparait pas dans le tableau) pour un ensemble de tableaux de differentes longeurs.

Question 5 : Dans le tableau `lengths` on a accumulé les entiers de `1` à `1000` ; pour chaque longueur `n` on crée un tableau `a` contenant les entiers de `0` à `n - 1` avec la fonction `list(range(n))`, qui convertit le `range` en tableau.

Mesurer le temps du calcul des deux fonctions `linear_search` et `binary_search` sur chaque tableau `a` dans le pire des cas (c'est-à-dire, en cherchant une valeure qui n'apparait pas) et ajouter les temps obtenus respectivement aux tableaux `times_linear` et `times_binary`.

Le code après la boucle `for` dessinnera un graphe `matplotlib` avec les temps de calcul mesurés.

Correction

```
from matplotlib.pyplot import plot, show, legend
from timeit import default_timer


#Q1

def linear_search(x, a):
    for i in range(len(a)):
        if a[i] == x:
            return i
    return -1
```

```
#Q2

print("Recherche linéaire:")

a1 = [5, 1, 3, 2, 7, 9, 4, 8, 0, 6]

print(linear_search(2, a1))
print(linear_search(5, a1))
print(linear_search(10, a1))
print(linear_search(6, a1))


#Q3

def binary_search(x, a):
    left = 0
    right = len(a) - 1
    while left <= right:
        m = (left + right) // 2
        if x == a[m]:
            # élément trouvé
            return m
        elif x < a[m]:
            # on cherche dans la moitié gauche
            right = m - 1
        else:
            # on cherche dans la moitié droite
            left = m + 1
    # élément absent
    return -1


#Q4

print("Recherche dichotomique:")

a2 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print(binary_search(2, a2))
print(binary_search(5, a2))
print(binary_search(10, a2))
print(binary_search(6, a2))
```

```
#Q5

# lengths = liste des longueurs des tableaux
lengths = []
for i in range(1, 1001):
    lengths.append(i)

times_linear = []
times_binary = []

for n in lengths:
    a = list(range(n))
    start = default_timer()
    result = linear_search(-1, a)
    end = default_timer()
    times_linear.append(end - start)
    start = default_timer()
    result = binary_search(-1, a)
    end = default_timer()
    times_binary.append(end - start)

if __name__ == '__main__':
    plot(lengths, times_linear, label="linear")
    plot(lengths, times_binary, label="binary")
    legend()
    show()
```

Partie 2 — Tri par sélection

Ex. 2

Il existe de nombreux algorithmes pour trier les tableaux (par exemple, le tri par insertion et le tri fusion qu'on a vu dans l'UE Introduction à l'informatique).

Le tri par sélection d'un tableau `a` consiste en chercher le minimum de `a`, en échanger cet élément avec l'élément en position `0` et en répéter cette procédure pour le sous-tableau de la position `1` à la position `n - 1`, puis pour le le sous-tableau de la position `2` à la position `n - 1`, etc., jusqu'à la fin du tableau.

Après l'`i`-ème étape de l'algorithme, la portion initiale du tableau (de la position `0` à la position `i`) sera trié, ce qui nous garantit que la totalité du tableau sera triè à la fin de l'algorithme.

Question 1 : Écrire la fonction `find_minimum(a, left, right)`, qui prend comme arguments un tableau `a` (de longueur `>= 1`) et deux entiers `left` et `right` et retourne la position de l'élément minimum du sous-tableau de la position `left` à la position `right`.

Question 2 : Affecter à la variable `m` le minimum du tableau `a1` avec la fonction `find_minimum`.

Question 3 : Écrire la fonction `swap(a, i, j)`, qui prend commme arguments un tableau `a` et deux entiers `i` et `j` et qui échange les éléments de position `i` et `j` en `a`. (La fonction ne retourne aucun résultat, puisque sa tâche est de modifier directement le tableau `a`.)

Question 4 : Échanger les éléments en position `1` et `7`, puis ceux en position `4` et `3` dans le tableau `a1` à l'aide de la fonction `swap`. Afficher le tableau `a1` ainsi modifié.

Question 5 : Écrire la fonction `selection_sort(a)`, qui prend comme argument un tableau `a` et le trie avec l'algorithme de tri par sélection. Utiliser les fonctions `find_minimum` et `swap` pour résoudre cet exercice. (La fonction ne retourne aucun résultat, puisque sa tâche est de modifier directement le tableau `a`.)

Question 6 : Trier le tableau `a1` à l'aide de la fonction `selection_sort` qu'on vient d'écrire. Afficher le tableau `a1` ainsi modifié.

Question 7 : Évaluer expérimentalement l'efficacité de l'algorithme de tri par sélection. Procéder comme dans le dernier exercice sur la recherche dans lex tableaux, en générant un tableau `a` de taille `n` pour chaque valeur de `n` dans un intervalle approprié (conseil : commencer avec un petit intervalle, par exemple de 1 à 100, puisque l'algorithme de tri est beaucoup plus lent que les algorithmes de recherche).

Utiliser la fonction `shuffle(a)` pour permuter de façon aléatoire le contenu du tableau `a`. Mesurer le temps de calcul de `selection_sort(a)` pour chaque tableau `a` et dessiner un graphe qui montre le temps pour chaque taille de tableau.

Après avoir analysé le graphe et le code, quelles conclusions peut-on tirer sur le temps de calcul de `selection_sort` en fonction de la taille des entrées ?

Correction

```
from matplotlib.pyplot import plot, legend, show
from timeit import default_timer
from random import shuffle
```

```
#Q1

def find_minimum(a, left, right):
    i = left
    for j in range(left + 1, right + 1):
        if a[j] < a[i]:
            i = j
    return i


#Q2

a1 = [9, 4, 1, 3, 0, 8, 2, 6, 7, 5]
print(a1)
m = find_minimum(a1, 0, len(a1)-1)
print("Le minimum est en position", m)
print()


#Q3

def swap(a, i, j):
    tmp = a[i]
    a[i] = a[j]
    a[j] = tmp


#Q4

print("Échange d'éléments")
swap(a1, 1, 7)
swap(a1, 4, 3)
print(a1)
print()

#Q5

def selection_sort(a):
    n = len(a)
    for i in range(n-1):
        m = find_minimum(a, i, n - 1)
        swap(a, i, m)


#Q6
```

```
print("Tri par selection")
selection_sort(a1)
print(a1)

#Q7

lengths = []
for i in range(1, 100):
    lengths.append(i)
times = []

for n in lengths:
    a = list(range(n))
    shuffle(a)
    start = default_timer()
    selection_sort(a)
    end = default_timer()
    times.append(end - start)

if __name__ == '__main__':
    plot(lengths, times)
    show()
```

TP8

Partie 1 — Récursivité

Ex. 1 — Factorielle

Consigne pour les tests automatiques

Dans la partie Récursivité du TP, un certain nombre de tests automatiques vous aideront à réaliser vos fonctions en vous donnant des indices si elles ne paraissent pas justes. Si vous voulez que ces tests fonctionnent le mieux possible et puissent vous indiquer la voie à suivre, vous devez nommer vos fonctions et leurs paramètres EXACTEMENT de la manière demandée dans l'énoncé.

Récursivité

En informatique, on dit qu'un algorithme est **récursif** quand il fait référence à lui-même.

Par exemple, la fonction suivante est récursive :

```
def print_n_hello(n):
    if n > 0:
        print("Hello")
    print_n_hello(n-1)
```

En effet, on trouve à l'intérieur de la définition de la fonction `print_n_hello(n)` un appel à cette même fonction. On appelle cela un **appel récursif**, c'est-à-dire l'utilisation d'une fonction dans sa définition.

Notre fonction permet donc d'afficher `n` fois le mot `"Hello"` en remarquant qu'afficher `n` fois ce mot revient à l'afficher une fois puis à recommencer pour `n-1`.

Quand on écrit une fonction récursive, il faut toujours penser à mettre un **cas de base**. Cela signifie qu'il faut savoir quand la fonction va arrêter de faire des appels récursifs, sinon, elle ne s'arrêtera jamais !

Par exemple, notre algorithme s'arrête car on rappelle la fonction à chaque fois avec un entier plus petit que lors de l'appel précédent (`n-1` au lieu de `n`) et quand `n` devient nul, on ne fait rien. On est donc certains qu'au bout d'un nombre d'appels récursifs fini, la fonction ne fera plus rien et s'arrêtera.

Questions

Certaines fonctions se définissent naturellement de manière récursive. Considérons la fonction factorielle : `n! = n x (n-1) x (n-2) x ... x 1`. On peut remarquer que `n! = n x (n-1)!` pour `n>1` et `1! = 1` ce qui constitue notre appel récursif et notre cas de base.

Question 1 : Définissez une fonction **récursive** `factorial(n)` qui renvoie le résultat du calcul `n!`.

Question 2 : Affichez le résultat de votre fonction pour `n=1`.

Question 3 : Affichez le résultat de votre fonction pour `n=10`.

Question 4 : Écrivez une nouvelle fonction `factorial_print_step(n)` qui calcule le résultat du calcul `n!` mais en plus affiche la valeur du paramètre `n` sur laquelle elle est appelée.

Question 5 : Affichez le résultat de votre fonction pour `n=10` et observez tous les appels récursifs à votre fonction.

Correction

```
# Q1
```

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)


# Q2
print(factorial(1))

# Q3
print(factorial(10))


# Q4
def factorial_print_step(n):
    print(n)
    if n == 1:
        return 1
    else:
        return n * factorial_print_step(n - 1)


# Q5
print(factorial_print_step(10))
```

Ex. 2 — Exponentiation

Quand une fonction ne fait pas un appel à elle-même, on dit qu'elle est **itérative**. Toutes les fonctions que vous avez écrites jusqu'à maintenant étaient donc itératives.

Exponentiation "normale"

Question 1 : Écrivez une fonction **itérative** `exponentiation(x, n)` pour calculer la valeur de x^n.

Question 2 : Affichez le résultat de votre fonction pour `x=2` et `n=4`.

Question 3 : Écrivez une fonction **récursive** `recursive_exponentiation(x, n)` pour calculer la valeur de x^n. On pourra remarquer que $x^n = x * x^{n-1}$.

Question 4 : Affichez le résultat de votre fonction pour `x=2` et `n=4`.

Exponentiation rapide

Lors d'un TP précédent, vous avez écrit une fonction d'exponentiation rapide. Pour cela, vous avez utilisé la remarque suivante pour calculer x^n : - si x est pair : $x^n = (x * x)^{n/2}$; - si x est impair : $x^n = x * (x * x)^{\frac{n-1}{2}}$.

Vous ne connaissiez pas la notion de fonction récursive à ce moment-là et vous avez donc écrit une fonction **itérative**.

Question 5 : Écrivez une fonction **récursive** `fast_exponentiation(x,n)` qui calcule x^n en utilisant la méthode de l'exponentiation rapide. Attention, n'oubliez pas les cas de base.

Question 6 : Affichez le résultat de votre fonction pour `x=2` et `n=4`.

Correction

```
# Q1
def exponentiation(x, n):
    result = 1
    for i in range(n):
        result = result * x
    return result


# Q2
print(exponentiation(2, 4))


# Q3
def recursive_exponentiation(x, n):
    if n == 0:
        return 1
    else:
        return x * recursive_exponentiation(x, n - 1)


# Q4
print(recursive_exponentiation(2, 4))


# Q5
def fast_exponentiation(x, n):
    if n == 0:
        return 1
```

```
    elif n == 1:
        return x
    else:
        if n % 2 == 0:
            return fast_exponentiation(x * x, n // 2)
        else:
            return x * fast_exponentiation(x * x, (n - 1) // 2)


# Q6
print(fast_exponentiation(2, 4))
```

Ex. 3 — Tri fusion

Le tri fusion est une méthode récursive de tri d'une liste. L'algorithme appelé pour trier une liste `l` se résume ainsi :

- On sépare la liste `l` en deux listes `l1` et `l2` de tailles égales.
- On trie `l1` et `l2` par la méthode du tri fusion.
- On fusionne les deux listes ainsi triées.

On vous donne la fonction `merge(l1, l2)` qui fusionne deux listes triées (troisième étape de l'algorithme).

Rappel : à partir d'une liste `l`, on peut créer une nouvelle liste `l1` contenant les `i` premiers éléments de `l` en utilisant `l1 = l[:i]`. On peut de même récupérer tous les éléments situés après l'indice `i` avec `l2 = l[i:]`.

Question 1 : Écrivez la fonction **récursive** `merge_sort(l)` qui utilise la fonction `merge` pour réaliser le tri fusion. N'oubliez pas **les** cas de base.

Question 2 : Affichez le résultat de votre fonction pour `[4, 7, 1, 0, 3, 6]`.

Correction

```
def merge(l1, l2):
    i1 = 0
    i2 = 0
    merged = []
    while i1 < len(l1) and i2 < len(l2):
        if l1[i1] < l2[i2]:
            merged.append(l1[i1])
            i1 += 1
        else:
```

```
            merged.append(l2[i2])
            i2 += 1

    while i1 < len(l1):
        merged.append(l1[i1])
        i1 += 1

    while i2 < len(l2):
        merged.append(l2[i2])
        i2 += 1

    return merged


# Q1
def merge_sort(l):
    if len(l) == 0 or len(l) == 1:
        return l
    else:
        n = len(l)
        l1 = l[:n // 2]
        l2 = l[n // 2:]
        l1_sorted = merge_sort(l1)
        l2_sorted = merge_sort(l2)
        return merge(l1_sorted, l2_sorted)


# Q2
print(merge_sort([4, 7, 1, 0, 3, 6]))
```

Ex. 4 — Fibonacci (Optionnel) : les possibles soucis du récursif

Vous avez vu lors de TPs précédents la définition de la suite de Fibonacci : $F(n) = F(n-1) + F(n-2), F(0) = 0, F(1) = 1$. Cette définition est naturellement récursive.

Question 1 : Écrivez une fonction **récursive** `recursive_fibo(n)` qui calcule le n^{ime} terme de la suite de Fibonacci et affiche lors de son appel la valeur de n sur laquelle elle a été appelée.

Question 2 : Affichez le résultat de votre fonction pour `n=7`. Que remarquez-vous ?

Cet exemple permet d'illustrer que même si les fonctions récursives sont souvent plus simples que leur version itérative, elles ne sont pas toujours plus efficaces.

Ici, en utilisant une fonction récursive qui implémente directement la définition de la suite, on va faire plusieurs fois exactement le même calcul et donc ne pas être efficace du tout.

Question 3 : Affichez le résultat de votre fonction pour `n=15`. Observez le nombre d'appels qui sont faits alors qu'on pourrait retourner le résultat en faisant seulement une quinzaine d'additions.

Correction

```
# Q1
def recursive_fibo(n):
    print(n)
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return recursive_fibo(n - 1) + recursive_fibo(n - 2)


# Q2
print(recursive_fibo(7))
# Q3
print(recursive_fibo(15))
```

Ex. 5 — Tours de Hanoi (Optionnel)

Le problème des tours de Hanoi est le suivant : - On dipose de trois piquets et de n palets de tailles $1, 2, 3, ..., n$. - Au départ, tous les palets sont empilés de manière décroissante sur le piquet de gauche : le palet de taille n est tout en bas et celui de taille 1 tout en haut. - On ne peut déplacer qu'un palet à la fois. - On ne peut déplacer que le palet le plus haut d'une pile. - On ne peut mettre un palet sur un autre que s'il est de taille plus petite. - On veut faire passer tous les palets sur le piquet de droite.

Une résolution récursive du problème pour plus de 1 palet est la suivante : - on bouge $n-1$ palets du piquet de départ au piquet intermédiaire; - on bouge le grand palet du piquet de départ au piquet d'arrivée; - on bouge $n-1$ palets du piquet intermédiaire au piquet d'arrivée.

Question 1 : Écrivez la fonction `hanoi(n, begin, end, temporary)` permettant de bouger n palets depuis le piquet `begin` vers le piquer `end` en utilisant le

piquet `temporary` comme piquet intermédiaire et qui renvoie les piquets begin, end, temporary modifiés. On représentera chaque piquet par une liste représentant les tailles des palets présents sur le piquet, de bas en haut.

Question 2 : Affichez le résultat de votre fonction appliquée au déplacement de 6 palets.

Correction

```
# Q1
def hanoi(n, begin, end, temporary):
    if n > 0:
        hanoi(n - 1, begin, temporary, end)
        element_to_move = begin.pop()
        end.append(element_to_move)
        hanoi(n - 1, temporary, end, begin)
    return begin, end, temporary


# Q2
print(hanoi(6, [6, 5, 4, 3, 2, 1], [], []))
```

Ex.6 — Fractales

Correction automatique

Attention, cet exercice vous demandant de faire des dessins, il n'y aura pas de tests automatiques. Lancez votre code et regardez si le dessin vous convient !

Turtle

Le module turtle de python permet la production facile de figures géométriques, à l'aide d'un ensemble d'instructions de dessins élémentaires.

On importe tout ce que contient un module par la ligne `from turtle import *`. Cela permet de dire à python de chercher le module `turtle` et de récupérer toutes (c'est le sens de *) les fonctions qui y sont définies.

Grâce à cette ligne, on a accès dans le fichier à toutes les fonctions de turtle : - `reset()` : efface le dessin - `up()` : relève le crayon pour ne plus dessiner - `down()` : abaisse le crayon pour dessiner - `forward(distance)` : avance d'une distance donnée - `backward(distance)`: recule d'une distance donnée - `left(angle)` :

tourne à gauche d'un angle exprimé en degrés - `right(angle)` : tourne à droite d'un angle exprimé en degrés.

Question 1 : Écrivez une fonction `draw_square(length)` qui dessine un carré de côté donné.

Question 2 : Appelez votre fonction pour dessiner un carré de côté 100.

Si vous souhaitez que votre dessin ne se ferme pas dès qu'il est terminé, vous pouvez utiliser le module `time` de python et sa fonction `sleep(time)` qui permet de faire une pause de `time` secondes, juste après le dessin. Il faut alors ajouter l'import de la fonction `sleep` depuis le module `time` : `from time import sleep` tout en haut du fichier, qui dit à python d'aller chercher la fonction qui nous intéresse dans le bon module.

Flocon de Von Koch

Le flocon de Von Koch est une fractale c'est-à-dire une construction géométrique récursive, invariante par changement d'échelle. Pour construire le flocon de Von Koch, on part d'un triangle équilatéral, puis, à chaque itération, on divise chaque côté du polygone en trois segments égaux. À partir de ces trois segments, on trace un triangle équilatéral orienté vers l'extérieur, de base le segment du milieu puis on supprime le segment du milieu.

On commencera par écrire une fonction qui permet d'appliquer un certain nombre de fois la transformation à un segment : - l'appliquer 0 fois trace le segment; - l'appliquer n fois revient à l'appliquer $n-1$ fois sur un segment de taille divisée par 3, tourner à gauche de 60 degrés, l'appliquer à nouveau $n-1$ fois sur un segment de taille divisée par 3, tourner à droite de 120 degrés, l'appliquer encore $n-1$ fois sur un segment de taille divisée par 3, tourner à gauche de 60 degrés et enfin l'appliquer $n-1$ fois sur un segment de taille divisée par 3.

Question 3 : Écrire une fonction `snowflake_side(length, steps)` qui permet d'appliquer `steps` fois la transformation à un segment de taille `length`.

Question 4 : Appelez votre fonction pour `2` étapes sur un segment de taille `100`.

Question 5 : Écrire la fonction `snowflake(length, step)` qui fabrique le flocon de Von Koch en appliquant `steps` fois la transformation pour un triangle équilatéral de départ de côté `length`.

Question 6 : Appelez votre fonction `snowflake` pour `1` étapes sur un segment de taille `250`.

Question 7 : Appelez votre fonction `snowflake` pour `2` étapes sur un segment de taille `250`.

Question 8 : Appelez votre fonction `snowflake` pour `5` étapes sur un segment de taille `250`. Vous pouvez augmenter la vitesse de dessin en utilisant

`speed(speed)` avec en paramètre la vitesse que vous souhaitez, par exemple 1000.

Correction

```
from turtle import *
from time import sleep


# Q1
def draw_square(length):
    for i in range(4):
        forward(length)
        right(90)


# Q2
draw_square(100)


# Q3
def snowflake_side(distance, step):
    if step == 0:
        forward(distance)
    else:
        snowflake_side(distance / 3, step - 1)
        left(60)
        snowflake_side(distance / 3, step - 1)
        right(120)
        snowflake_side(distance / 3, step - 1)
        left(60)
        snowflake_side(distance / 3, step - 1)


# Q4
reset()
snowflake_side(100, 2)


# Q5
def snowflake(distance, etape):
    for i in range(3):
        snowflake_side(distance, etape)
        right(120)
```

```
# Q6
reset()
speed(10)
snowflake(250, 1)
sleep(5)

# Q7
reset()
speed(10)
snowflake(250, 2)
sleep(5)

# Q8
reset()
speed(1000)
snowflake(250, 5)
sleep(5)
```

Partie 2 — Arbres

Ex. 7 — Expressions

Cet exercice se fera dans la console.

Types

Jusqu'à present nous avons utilisé des types simples, comme par exemple les entiers et les chaines de caractères.

Question 1 : Evaluer dans la console :

```
>>> type(2)
>>> type("2")
>>> type(int("2"))
>>> type(str(2))
```

Dans le premier cas la console renvoie `<class 'int'>` *pour signifier que* `2` *est du type simple* `int`*. Savez-vous pourquoi le second cas renvoie* `str`*? Comment expliquer les deux derniers cas?*

Nous avons aussi utilisé des types composés, comme par exemple les listes:

Question 2 : Evaluer dans la console :

```
>>> type([1,2,3])
```

Nous n'avions pas de type pour représenter par exemple le symbole d'une inconnue x, dans une expression mathématique comme $2 + x$.

Question 3 : Evaluer dans la console :

```
>>> 2+x
>>> 2+"x"
```

Notez que dans un premier temps l'interpréteur se plaint de ne pas connaître la variable x, ce qui cause une erreur. En effet: une variable au sens informatique du terme, ce n'est pas la même chose qu'une inconnue au sens mathématique du terme. Une variable, en informatique, c'est un emplacement mémoire, qui contient une valeur concrète. Une inconnue, en mathémétique, est un symbole qui n'a pas de valeur concrète associée.

Dans une second temps nous avons tenté de représenter l'inconnue x par une simple chaîne de caractère, ce est une bonne idée. Le problème, c'est que python ne comprends pas que l'on veuille sommer des chaînes de caractères et des entiers. Cela cause une erreur de type.

Nous allons utiliser le module `sympy` pour avoir un type symbole inconnu.

Question 4 : Evaluer dans la console :

```
>>> from sympy import *
>>> S("x")
>>> type(S("x"))
>>> 2+S("x")
```

Le module `sympy` nous a donné une fonction `S(.)` qui crée des symboles. Ces symboles sont d'un type particulier, définit par `sympy`. On peut y songer comme à des chaînes de caractères, mais qui cette fois admettent d'être additionnées à des entiers. Ces symboles vont nous permettre de representer les inconnues en mathématiques.

Calcul symbolique

Nous savions déjà que nous pouvions utiliser la console python comme un calculatrice. Maintenant, grâce à `sympy`, nous pouvons nous en servir pour faire de l'algèbre.

Tout d'abord ameillorons l'affichage de nos expressions.

Question 5 : Evaluer dans la console :

```
>>> x=S("x")
>>> (sqrt(2)+x**2)/x
>>> init_printing()
>>> (sqrt(2)+x**2)/x
```

Notez que la première ligne met le symbole x dans une variable que l'on a appelé `x`*, mais que l'on aurait pu appeler autrement. Cela nous permet juste d'écrire* `x` *au lieu de* `S("x")`*. Notez comme la procédure* `init_printing()` *met en place un affichage plus mathématique de nos expressions.*

Puis, découvrons quelques fonctionnalités de `sympy`.

Question 6 : Evaluer dans la console :

```
>>> diff(sin(x)+x**2)
>>> integrate(sin(x)+x**2)
>>> solve(Eq(x**2+2,0), x)
>>> simplify(cos(x)**2+sin(x)**2)
>>> expand((x+1)**10)
>>> factor(x**2+100+20*x)
```

Ces fonctionnalités sont présentes aussi dans tout Computer Algebra System (CAS) qui se réspecte, comme Mathematica, Maple, Sage, Matlab… Elles couvrent probablement la totalité du programme de Licence 1 de Mathématiques, et au-delà. Cela signifie, donc, que derrière chacune des notions qui vous sont enseignées actuellement en mathématiques, se cache un algorithme, que l'on peut expliciter et coder notamment en python.

Correction

```
# TODO: Cet exercice est à faire dans la console python.
```

Ex. 8 — Arbres

Cet exercice se fera dans la console.

Pour cet exercice on suppose que vous avez déjà évalué:

```
>>> from sympy import *
>>> x=S("x")
>>> init_printing()
```

Au cours de l'exercice précédent nous avons vu quel était le type de `x` ou de `2` dans l'expression `2+x`, en évaluant:

```
>>> type(x)
>>> type(2)
```

Mais, quel est le type d'une expression comme `2+x`?

Question 1 : Evaluer dans la console :

```
>>> type(2+x)
>>> type(2*x)
```

Notez que `2+x` est de type `Add`, alors que `2*x` est de type `Mul`.

Sympy a donc définit un type `Add` pour représenter les sommes, un type `Mul` pour représenter les produits, etc. On devine que type `Add` est un type composé, tout comme l'était le type `List`, puisqu'une expression de type somme doit contenir la liste des éléments sommés.

Question 2 : Evaluer dans la console :

```
>>> expr=2+x
>>> expr
>>> expr.func
>>> expr.args
>>> expr.args[0]
>>> expr.args[1]
```

Notez que `(2+x).args` renvoie la paire `(2,x)` des éléments sommés.

De même qu'une liste peut contenir des sous listes, une expression peut contenir des sous-expressions.

Question 3 : Evaluer dans la console :

```
>>> expr=x*(2+x)
>>> expr
>>> expr.func
>>> expr.args
>>> expr.args[1]
>>> expr.args[1].func
>>> srepr(expr)
```

Notez que `(x*(2+x)).args[1]` n'est autre que l'expression `2+x`.

Les types composés sont aussi appelés "structures de données" par les informaticiens. Ici, la structure de donnée utilisée pour par `sympy` pour représenter les expressions peut être comparée à un arbre. En effet, concernant l'expression `expr=x*(2+x)`: * La racine de l'arbre est une multiplication. C'est que renvoie `expr.func`. * L'arbre contient deux sous arbres. C'est ce que renvoie `expr.args`. * Le premier sous-arbre est le symbole x. C'est ce que renvoie `expr.args[0]`. Il ne contient aucun sous-arbre, on dit c'est une feuille. * Le deuxième sous-arbre est l'expression $2 + x$. C'est ce que renvoie `expr.args[1]`. * La racine du sous-arbre est une somme. Le sous-arbre contient deux sous-sous-arbres. C'est ce que renvoie `expr.args[1].args`. * Les deux sous arbres sont des feuilles: l'une est le symbole x, l'autre l'entier 2.

D'un manière générale, une structure de donnée de type arbre est un type défini récursivement, comme étant: * soit une feuille (cas de base); * soit un arbre ayant une racine est des sous-arbres (cas récursif).

Ces structures de données sont extrêmement utilisées en informatique, par exemple pour représenter des programmes, des hiérarchies, et bien sûr l'arborescence

de vos fichiers (répertoires contenant des sous-répertoires, etc.).

Correction

```
# TODO: Cet exercice est à faire dans la console python.
```

Ex. 9 — Parcours

Cet exercice se fait dans `task.py`.

Dans la première partie de ce TP nous avons étudié les algorithmes récursifs. Dans l'exercice précédent nous avons étudié les structures de données définies récursivement : les arbres. Ici, nous allons appliquer un algorithme récursif, à une structure de donnée récursive: nous allons faire un parcours d'arbre.

Aplatissement d'abre

La fonction `flatten` a pour but de parcourir et afficher un arbre d'expression `sympy`. Par exemple `flatten(2+x)` affiche:

```
addition de :
    2
    x
```

Question 1 : Compléter `flatten` afin qu'elle puisse gérer la multiplication.

Question 2 : Appliquer `flatten` sur l'expression $x(2+x)$.

Evaluation d'une expression

La fonction `calculate` prend en données une expression, et une valeur pour x, et a pour but de calculer numériquement la valeur de l'expression en ce point. Par exemple `calculate(2+x,8)` renvoie `10` en sortie.

Question 3 : Compléter `calculate` afin qu'elle puisse gérer la multiplication.

Question 4 : Appliquer `calculate` sur l'expression $x(2+x)$, pour $x = 7$ et afficher la valeur de retour.

Correction

```
from sympy import *
```

```
init_printing()
x = S("x")
# Ignore the next four lines if you want, they are just a trick
# to recover the names of the different sympy types once and for all
typeofsymbol = x.func
typeofinteger = S("2").func
typeofaddition = (2 + x).func
typeofmultiplication = (2 * x).func


# Q1
def flatten(expr):
    iflatten("", expr)


def iflatten(indent, expr):
    # This is the base case
    if expr.func == typeofsymbol or expr.func == typeofinteger:
        print(indent, expr)
    # Those are the recursive case
    elif expr.func == typeofaddition:
        print(indent, "addition de :")
        iflatten(indent + "    ", expr.args[0])
        iflatten(indent + "    ", expr.args[1])
    elif expr.func == typeofmultiplication:
        print(indent, "multiplication de :")
        iflatten(indent + "    ", expr.args[0])
        iflatten(indent + "    ", expr.args[1])
    # We are only handling a sublanguage of sympy
    else:
        print(indent, "Expression inconnue par flatten.")


# Q2
flatten(x * (2 + x))


# Q3
def calculate(expr, val):
    # This is the base case
    if expr.func == typeofsymbol:
        return val
    if expr.func == typeofinteger:
        return int(expr)
    # Those are the recursive case
```

```
    elif expr.func == typeofaddition:
        return calculate(expr.args[0], val) + calculate(expr.args[1], val)
    elif expr.func == typeofmultiplication:
        return calculate(expr.args[0], val) * calculate(expr.args[1], val)
    # We are only handling a sublanguage of sympy
    else:
        print("Expression inconnue par calculate.")
        return float("NaN")


# Q4
print(calculate(x * (2 + x), 7))
```

TP9

Partie 1 — Cryptographie

Ex. 1 — Codage de César

Le Codage de César est un procédé de cryptographie permettant d'encoder en message en chiffrant chaque lettre du message en une autre lettre, décalée dans l'ordre alphabétique par un nombre de caractères connu à l'avance. On appelle ce nombre de caractères la clé d'encryption du message.

L'objectif de cet exercice est de créer des fonctions d'encodage et de décodage suivant la méthode du codage de César.

Dans le détail, le codage de César est basé sur la substitution des caractères suivant leur position dans l'ordre alphabétique.

Prenons comme exemple la clé `n=3` . Avec cette clé, chaque lettre non accentuée devient après encodage la lettre située trois positions après dans l'ordre alphabétique.

Ainsi `'a'` devient `'d'`, `'f'` (la sixième lettre de l'alphabet) devient `'i'` (la neuvième lettre) et `'x'` (24ème) devient `'a'` (car `1 = (24 + 3) % 26`).

Le codage fonctionne également avec une clé négative. Il faut alors reculer dans l'ordre alphabétique, tout en conservant le côté cyclique du codage. `'D'` devient `'Y'` avec la clé de cryptage `-5`.

Pour encoder un message complet (de plusieurs caractères) avec une certaine clé, - chaque lettre non accentué minuscule devient la lettre minuscule décalée par la clé - chaque lettre non accentué majuscule devient la lettre majuscule décalée par la clé - tous les autres caractères (lettre accentuées, ponctuation, espaces) restent identiques

Question 1 : **Écrivez** le code de la fonction `code_char(c,cle)`, qui **renvoie** le caractère correspondant à l'encodage de César du caractère `c` par la clé `cle`.

Pour ce faire, il est pratique d'utiliser les fonctions `ord` et `chr`, permettant de transformer un caractère en un entier le représentant (et respectivement un entier en le caractère correspondant). Dans la console, essayez les commandes suivantes et déduisez-en comment encoder un caractère.

```
ord('a'), ord('d'), ord('z')
ord('a') - ord('a'), ord('d') - ord('a'), ord('z') - ord('a')
ord('A'), ord('Z')
ord(' '), ord('.')
chr(ord('G'))
chr(ord('a') + 5)
```

Remarque: une des propriétés importantes de ord est d'avoir les 26 lettres minuscules représentés par des entiers consécutifs. C'est également le cas pour les lettres majuscules.

Question 2 : **Écrivez** la fonction `encode(s,cle)` qui **renvoie** la chaîne de caractère `s` encodée avec la clé `cle`. Pour ce faire, créez une autre chaîne de caractère initialement vide et ajoutez les caractères encodés un par un.

Question 3 : **Écrivez** la fonction `decode(s,cle)` qui décode une chaîne `s` encodée avec la clé `cle`, et **renvoie** le message décodé.

Question 4 : À l'aide de la fonction `decode`, trouvez quel est la clé décodant le message `"Sbrl, ql zbpz avu wèyl."`. Pour cela, vous pouvez par exemple tester toutes les clés possibles, et lire le résultat de chaque tentative de décodage. Seul l'un d'entre eux fait sens.

Correction

```
#Q1
def code_char(c,cle):
    n = ord(c)
    if(n >= ord('a') and n <= ord('z')):
        return chr(ord('a') + ((n - ord('a') + cle) % 26))
    elif(n >= ord('A') and n <= ord('Z')):
        return chr(ord('A') + ((n - ord('A') + cle) % 26))
    else:
        return c


def print_code_char(c,cle):
    print("\'" + c + "\' est codé en \'" + code_char(c,cle) + "\' avec la clé " + str(cle))
```

```
print("Question 1:")
print_code_char('d',5)    #doit transformer 'd' en 'i'
print_code_char('i',-15)  #doit transformer 'i' en 't'
print_code_char('P',20)   #doit transformer 'P' en 'J'
print_code_char(' ',5)    #doit transformer ' ' en ' '

#Q2

def encode(s,cle):
    s2 = ""
    for i in range(len(s)):
        s2 += code_char(s[i],cle)
    return s2

print()
print("Question 2:")
print(encode("La vie est un long fleuve tranquille.",3))
print(encode("La vie est un long fleuve tranquille.",11))
print(encode("La vie est un long fleuve tranquille.",-7))

#Q3
def decode(s,cle):
    return encode(s,-cle)

print()
print("Question 3:")
print(decode("Wl gtp pde fy wzyr qwpfgp eclybftwwp.",-15))

#Q4
message_a_decoder = "Sbrl, ql zbpz avu wèyl."
cle_decodage = -7

print()
print("Question 4:")
print(encode(message_a_decoder,cle_decodage))
```

Ex. 2 : Codage Vigenère

Le Codage de Vigenère est un autre encodage substituant chaque caractère d'un message par un autre afin de l'encrypter.

Plutôt que de toujours utiliser la même méthode d'encodage des caractères, il

est basé sur l'utilisation d'une clé sous la forme d'une chaîne de caractères (avec uniquement des lettres minuscules).

Chaque lettre du message se voit associer un caractère de la chaîne `cle` suivant le nombre de lettres (non accentuées) qui le précède: - la première lettre du message se voit associer la première lettre de la clé - la seconde lettre du message se voit associer la seconde lettre de la clé - la troisième lettre du message se voit associer la troisième lettre de la clé - … - Dès que l'on a fini d'associer toutes les lettres de la clé, on recommence à associer les lettres du début de la clé de manière cyclique.

Ensuite, on effectue un encodage de chaque caractère `c` avec un décalage dans l'ordre alphabétique correspondant au même décalage transformant `'a'` en la lettre de la clé associée à `c`.

Ainsi, avec la clé `"arbre"`, on obtient les lettres associés, et les encodages suivants:

```
Message       : "Ceci est un message a encoder."
Cles associées: "arbr ear br earbrea r brearbr "
Message codé  : "Cvdz isk ve qejtrke r fegoufi."
```

- Le `'C'` est associé à `'a'` et donc est décalé de `0`, pour donner `'C'`.
- Le `'e'` est associé à `'r'` et donc est décalé de `14`, pour donner `'v'`.
- Le `'c'` est associé à `'b'` et donc est décalé de `1`, pour donner `'d'`.

Question 1 : **Copiez** (ou réécrivez) la fonction `code_char` utilisée pour le codage de César dans le premier exercice.

Question 2 : **Écrivez** la fonction d'encodage de Vigenère, `encode(s,cle)`, qui encode `s` avec la chaîne de caractère `cle`. Attention à bien différencier les lettres (faisant avancer dans la clé) des autres caractères.

Question 3 : **Écrivez** la fonction de décodage de Vigenère, `encode(s,cle)`, qui décode une chaîne codée `s` avec la chaîne de caractère `cle`. Il faut alors faire l'opération inverse de l'encodage pour chaque caractère.

Correction

```
#Q1 -- reprendre code_char du codage de César.

def code_char(c,cle):
    n = ord(c)
    if(n >= ord('a') and n <= ord('z')):
        return chr(ord('a') + ((n - ord('a') + cle) % 26))
    elif(n >= ord('A') and n <= ord('Z')):
        return chr(ord('A') + ((n - ord('A') + cle) % 26))
```

```
    else:
        return c

#Q2
def encode(s,cle):
    s2 = ""
    j = 0
    for i in range(len(s)):
        if((s[i] >= 'a' and s[i] <= 'z') or (s[i] >= 'A' and s[i] <= 'Z')):
            s2 += code_char(s[i],ord(cle[j]) - ord('a'))
            j = (j + 1) % len(cle)
        else:
            s2 += s[i]
    return s2

print()
print("Question 2:")
print(encode("Ceci est un message a encoder.","arbre"))  #l'encodage attendu est "Cvdz isk v
print(encode("Peu lui importe de quoi demain sera fait","petitfrere")) #l'encodage attendu e


#Q3
def decode(s,cle):
    s2 = ""
    j = 0
    for i in range(len(s)):
        if((s[i] >= 'a' and s[i] <= 'z') or (s[i] >= 'A' and s[i] <= 'Z')):
            s2 += code_char(s[i],ord('a') - ord(cle[j]))
            j = (j + 1) % len(cle)
        else:
            s2 += s[i]
    return s2

print()
print("Question 3:")
print(decode("Diepe dorr n'owg naj k vrnnvr","kenyarkana"))
```

Ex. 3 : Codage spartiate

Le codage Spartiate est une méthode d'encryption par transposition. Contrairement à César ou Vigenère, les caractères sont tous conservés, mais déplacés les uns par rapport aux autres rendant le message irreconnaissable.

Ce codage nécessite une clé de transposition `n`, qui est un nombre entier positif. À partir de cette clé, un message à coder est découpé en `n` segments de même longueurs (on ajoute des espace à la fin du message autant que nécessaire). Ensuite on construit le message codé de la façon suivante:

- Les `n` premières caractères du message codé sont:
  - Le premier caractère du segment n°`1`
  - Le premier caractère du segment n°`2`
  - …
  - Le premier caractère du segment n°`n`
- Les `n` caractères suivants sont:
  - Le second caractère du segment n°`1`
  - Le second caractère du segment n°`2`
  - …
  - Le second caractère du segment n°`n`
- Ainsi de suite jusqu'à avoir utilisé tous les caractères des segments.

Une façon de représenter ce codage est d'observer la matrice de `n` lignes et `ceil(len(s)/n)` colonnes, avec ceil la fonction donnant la partie entière supérieure d'un nombre réel. On écrit ensuite le message, un caractère par case. Le message codé est alors la suite des caractères écrits colonne par colonne.

Prenons l'exemple de `s="La vie c'est comme une boîte de chocolats, on ne sait jamais sur quoi on va tomber."`, avec `n=6`:

```
La␣vie␣c'est␣c
omme␣une␣boîte
␣de␣chocolats,
␣on␣ne␣sait␣ja
mais␣sur␣quoi␣
on␣va␣tomber.␣
```

Le message codé est alors `Lo  moamdoan meni ve  svi cn aeuhes  no utcecsro' oa mebliqbsoatuetît or tsji.ce,a`.

Question 1 : **Écrivez** la fonction `encode(s,n)`, qui renvoie `s` encodée par la méthode spartiate avec la clé `n`.

Vous pouvez utiliser la fonction `ceil` de la bibliothèque math, qui donne la partie entière arrondie au supérieur de son argument (si jamais vous avez besoin de la partie inférieur, la fonction `floor` existe.), notamment pour calculer à l'avance le nombre de caractères par ligne de la matrice.

Question 2 : **Écrivez** la fonction `decode(s,n)`, qui renvoie la chaîne décodée, en supposant que le code utilisé est la méthode spartiate avec la clé `n`.

Question 3 : **Trouvez** la clé permettant de décoder le message `"L n eq.ae f u  sllti vtoerl i nual eugvne "`, si on vous assure qu'un message en français a été encodé avec le codage spartiate.

Correction

```
from math import *

#Q1

def encode(s,n):
    s2 = ""
    m = ceil(len(s) / n)
    for i in range(m):
        for j in range(n):
            if i + j * m < len(s):
                s2 += s[i + j * m]
            else:
                s2 += " "
    return s2

print("Question 1:")
print(encode("La vie c'est comme une boîte de chocolats, on ne sait jamais sur quoi on va to
## Résultat attendu: "Lo  moamdoan meni ve  svi cn aeuhes  no utcecsro' oa mebliqbsoatuetît


#Q2
def decode(s,n):
    return encode(s,ceil(len(s)/n))

print()
print("Question 2:")
print(decode("Càq s' up emea?so r titl   ue ",5))
print(decode("L, snru s  ekjutp.eeioè ",6))

#Q3
message_a_decoder = "L n eq.ae f u  sllti vtoerl i nual eugvne "
cle_de_decalage = 7

print()
print("Question 3:")
print(decode(message_a_decoder,cle_de_decalage))
```

Partie 2 — Autoreproduction

Ex. 4 — Les fichiers

Fichiers

L'informatique pourrait être définie comme la science du stockage, de la transmission, et du traitement de l'information. Cette information peut provenir de différentes sources.

Pour un programme, l'information peut provenir par exemple : * de l'utilisateur (cf. l'instruction `input()` en python) ; * d'un sous-programme (cf. l'instruction `return` d'une fonction python) * du réseau ; * d'un fichier…

Au cours de cet exercice nous allons apprendre à lire un fichier, puis écrire dans un fichier, après avoir manipulé son contenu.

Lecture

Pour lire un fichier appelé `nomdufichier.type` il faut

1. Ouvrir le fichier en lecture: `f=open("nomdufichier.type","r")`.
2. Lire le contenu du fichier: `contenu=f.read()`.
3. Fermer le fichier: `f.close()`.

La variable `f` contient un "handler" vers le fichier ouvert. C'est-à-dire toutes les informations dont votre programme a besoin pour manipuler fichier. On peut y penser comme étant le "petit nom" de `nomdufichier.type`, en qui concerne le programme.

Question 1 : Tenter d'ouvrir un fichier inexistant. Qu'observez vous? Ouvrir le fichier `message.txt` et afficher son contenu.

Manipulation

En 2060, l'Académie Française décide de suivre l'exemple de la "Real academia española". Comme c'est le cas depuis longtemps pour l'espagnol, l'orthographe de la langue française sera donc simplifié au maximum. L'objectif, à terme, est d'obtenir un orthographe phonétique. Toutefois, afin de ne pas choquer la frange la plus conservatrice de la population, seul un premier volet de la réforme est introduit.

Voici la liste officielle des transformations à appliquer: * "er", "ez" -> "é" * "eau", "au", "ô" -> o * "ill" -> "y" * "ge" -> "j"

Vous êtes chargé de moderniser l'orthographe des "Fleurs du mal" de Charles Baudelaire. Pour vous aider, sacher que si `contenu` est une chaine de caractère, alors `contenu.replace("truc","machin")` en est une autre, obtenue en replaçant dans `contenu` toutes les occurences de "truc" par "machin".

Question 2 : Au lieu d'ouvrir `message.txt` comme à la question précédente, ouvrez `fleursdumal.txt`. Appliquez une série d'opérations de la forme `contenu=contenu.replace(...)` afin d'accomplir la transformation. Affichez le résultat.

Ecriture

Pour écrire dans un fichier appelé `nomdufichier.type` il faut

1. Ouvrir le fichier en lecture: `g=open("unautrefichier.type","r")`.
2. Ecrire une chaîne de caractère dans le fichier: `g.write(contenu)`.
3. Fermer le fichier: `g.close()`.

Question 3 : Ecrire dans un fichier `fleursdumal-moderne.txt` le résultat de la manipulation précédente. Lire par exemple le poème "La géante" pour vérifier le résultat.

Correction

```
#Q1
f=open("message.txt","r")
contenu=f.read()
f.close()
print(contenu)

#Q2
f=open("fleursdumal.txt","r")
contenu=f.read()
f.close()
contenu=contenu.replace("er ","é ")
contenu=contenu.replace("ez ","é ")
contenu=contenu.replace("eau","o")
contenu=contenu.replace("au","o")
contenu=contenu.replace("ô","o")
contenu=contenu.replace("ill","y")
contenu=contenu.replace("ge","je")
print(contenu)

#Q3
g=open("fleursdumal-moderne.txt","w")
g.write(contenu)
g.close()
```

Ex. 5 — Commandes systèmes

De tous les programmes qui s'exécutent sur votre ordinateur, le plus important est le système d'exploitation ("Operating System") : MacOs, Linux, Windows. Techniquement, c'est en fait un ensemble de programmes qui agissent de concert pour fournir toutes les fonctionnalités de bases : gestion du microprocesseur, de la mémoire, des fichiers, des périphériques, etc.

Grâce aux modules `os` et `sys`, vos programmes `python` peuvent intéragir avec le système d'exploitation, en lui envoyant des commandes et en observant ce qu'il renvoie. Par exemple l'instruction

```
system("commande")
```

demande au système d'exploitation d'exécuter la commande en question.

Question 1 : Demander au système d'exploitation d'exécuter la commande `pwd`.

`pwd` signifie "print working directory". La commande affiche l'endroit, dans le système de fichier entier de l'ordinateur, dans lequel vous êtes en train de travailler actuellement.

Question 2 : Afficher la valeur `argv[0]`. Que contient-elle?

`argv[0]` contient le nom entier du programme courant. Si le programme est appelé avec un argument, `argv[1]` contient cet argument. C'est la manière dont le système d'exploitation informe le programme des conditions dans lesquelles il a été exécuté.

Question 3 : Demander au système d'exécuter un autre programme `./myscript.py`. Qu'observez-vous?

Sous Linux du moins, vous observerez que ce fichier n'a pas le droit d'être exécuté.

Question 4 : Demander au système de vous donner les droits en exécution pour `./myscript.py`, avec la commande `chmod u+x myscript.py`. Demander au système d'exécuter le programme `./myscript.py`. Qu'observez-vous? Etudier le code de `myscript.py`. A quoi sert la première ligne?

Sous Linux du moins, la première ligne de `myscript.py` sert à spécifier que le texte contenu dans `myscript.py` n'est autre qu'un programme `python`.

Correction

```
import os, sys
```

```
#Q1
os.system("pwd")

#Q2
print(sys.argv[0])

#Q3
os.system("./myscript.py")

#Q4
os.system("chmod u+x myscript.py")
os.system("./myscript.py")
```

Ex. 6 — Vie

Dixit wikipedia. En biologie, une entité est traditionnellement considérée comme vivante si elle présente les activités suivantes, au moins une fois durant son existence : 1. Développement ou croissance : l'entité grandit ou mûrit jusqu'au moment où elle devient capable de se reproduire ; 2. Métabolisme : consommation, transformation et stockage d'énergie ou de masse; croissance en absorbant de l'énergie ou des nutriments présents dans son environnement ou en réorganisant sa masse, par production d'énergie, de travail et rejet de déchets ; 3. Motilité externe (locomotion) ou interne (circulation) ; 4. Reproduction : pouvoir créer de façon autonome d'autres entités similaires à soi-même. 5. Réponse à des stimuli : pouvoir détecter des propriétés de son environnement et d'agir de façon adaptée.

Question 1 : En combinant les réponses des exercices précédents, écrire quatre lignes de code qui fassent que votre programme: 1. Met son propore nom dans une variable `me`. 2. Lit le contenu de son propre fichier 3. Affiche ce contenu

Au danemark il fut courant que Christian appelle son fils Christiansen, Peder appelle son fils Pedersen, etc.

Question 2 : Créer une variable `son` dont le contenu est la chaine caractère `me`, où `".py"` a été remplacé par `"sen.py"`.

Question 3 : En combinant les réponses des exercices précédents, écrire trois lignes de code qui fassent que le contenu de votre programme s'écrive dans un fichier ayant comme nom celui retenu par la variable `son`.

Question 4 : Demander au système de donner les droits en exécution au fichier créé à la question précédente.

Question 5 : Demander au système d'exécuter le fichier créé à la question précédente. Avant de lancer l'exécution de votre programme, ouvrez le répertoire courant dans un explorateur de fichier, et apprêtez vous à interrompre le

processus avec `Crtl+C`. Lancez. Qu'observez-vous dans l'explorateur de fichier? Pourquoi? A quoi sert la ligne `time.sleep(1)` qui précède?

Correction

```
#!/usr/bin/python

import sys,os,time

#Q1
me=sys.argv[0]
f=open(me,"r")
contenu=f.read()
f.close()
print(contenu)

#Q2
son=me.replace(".py","sen.py")
print(son)

#Q3
g=open(son,"w")
g.write(contenu)
g.close()

#Q4
os.system("chmod u+x "+son)


time.sleep(1)

#Q5
os.system(son)
```

TP10

Partie 1 — Graphes

Ex. 1 — Création

Avant de commencer

Vérifier que votre interpréteur dispose bien des librairies `matplotlib` et `networkx` pour ce TP. Dans `Pycharme > File > Settings > Project TPx > Intepreter` vérifier s'ils apparaissent dans la liste, sinon cliquer sur le `+` et installer.

IMPORTANT : Dans ce TP, vous allez parfois devoir dessiner des graphes, ce qui ouvre une fenêtre supplémentaire avec le dessin du graphe demandé. Les fenêtres s'ouvrent une à la fois et il faut donc les fermer une par une pour que le code continue de s'éxécuter. Pendant les corrections automatiques, PyCharm exécute vos fichiers et il risque donc d'ouvrir énormément de fenêtres que vous devrez fermer une par une. Pour éviter cela : - quand on vous demande de dessiner un graphe et que vous voulez voir le résultat, tapez votre ligne de code normalement et **EXÉCUTEZ** le fichier (en faisant Clic Droit/Run) pour voir votre dessin; - quand vous voulez utiliser la correction automatique, **COMMENTEZ** (c'est-à-dire ajoutez un # au début de la ligne) toutes les lignes qui font des dessins (la correction automatique vous comptera quand même juste).

Graphes

Les graphes sont utilisés pour résoudre divers problèmes. Ce sont des modèles abstraits de dessins de réseaux reliant des objets et leurs applications sont nombreuses dans tous les domaines liés à la notion de réseau (réseau social, réseau informatique, télécommunications, etc.) et dans bien d'autres domaines (par exemple génétique).

Un graphe est un ensemble de points nommés nœuds (parfois sommets ou cellules) reliés par des traits (segments) ou flèches nommées arêtes (ou liens ou arcs).

On note $G = (V, E)$ un graphe G composé d'un ensemble de noeuds V et un ensemble d'arêtes E.

Implémentation par matrice d'adjacence

Rappel sur les listes. Vous avez vu lors du TP5 comment manipuler des listes. Pour accéder à l'élément d'indice i d'une liste L, on écrit `L[i]`.

Ici, nous allons manipuler des listes de listes. Par exemple, si `L = [[1,2], [3,4]]`. Dans ce cas, `L[0] = [1,2]`.

On peut donc accéder aux éléments de `L[0]` en faisant par exemple `L[0][1]` qui est égal à 2.

Matrice d'adjacence. * Une approche pour représenter un graphe est la matrice d'adjacence. L'idée ici est de construire une matrice de taille $n \times n$ où n est le nombre de noeuds. On numérote ensuie les noeuds de 0 à n-1, correspondant aux indices de la liste. La matrice contient les valeurs 1 ou 0: - la valeur 1 dans la case du tableau d'indice i,j indique la présence d'une arête entre les noeuds i et j; - la valeur 0 indique l'absence d'arête.

- La matrice d'adjacence ci-dessous représente donc un graphe :

```
G=[[0, 1, 1, 0, 0],
   [1, 0, 1, 1, 0],
   [1, 1, 0, 1, 1],
   [0, 1, 1, 0, 0],
   [0, 0, 1, 0, 0]]
```

`G[0][1] = 1` donc les sommets 0 et 1 sont voisins.

A l'inverse, `G[0][3] = 0` ce qui signifie que les sommets 0 et 3 ne sont pas voisins.

Terminologie

- Noeud : généralement représenté par un point dans un graphe. On les numérote $0, 1, 2, ..., n-1$
- Arête : une connexion entre deux noeuds. La ligne connectant 0 et 1 est un exemple d'arête.
- Boucle : quand une arête a pour extrémités le même sommet.
- Degré d'un noeud : le nombre d'arêtes/arcs qui lui sont incidents. Le degré du noeud 2 est 4 : il a une arête vers 0, vers 1, vers 3 et vers 4.
- Adjacence : connexion(s) entre un noeud et ses voisins. Le noeud 2 est adjacent au noeud 4 car il y a une arête qui lie les deux noeuds.

Exercice

La fonction `randint(a,b)` permet de générer des nombres entiers aléatoires entre `a` et `b`.

Question 1 : Écrivez une fonction `random_row(n)` qui crée une liste de taille `n` contenant des `0` et des `1` de manière aléatoire.

Question 2 : Essayez votre fonction pour créer une liste de `6` éléments.

Question 3 : Écrivez une fonction `random_oriented_graph(n)` qui génère un graphe aléatoire contenant `n` sommets.

Question 4 : Affichez un graphe orienté aléatoire de 4 sommets généré par votre fonction.

Dans le cas d'un graphe non orienté, la matrice d'adjacence est symétrique. En effet, une arête entre i et j est aussi une arête entre j et i. On donne la fonction `symetrize(M)` qui permet de modifier la matrice donnée pour la rendre symétrique en conservant sa partie triangulaire supérieure.

On considèrera de plus que nos graphes n'ont pas de boucles, ce qui signifie que la diagonale de la matrice d'adjacence doit ne contenir que des 0.

Question 5 : Écrivez une fonction `random_graph(n)` qui génère un graphe non orienté contenant `n` sommets. Pour cela, générez un graphe orienté quelconque avec votre fonction `random_oriented_graph`, mettez des 0 sur la diagonale puis rendez-là symétrique en utilisant la fonction `symetrize`.

Question 6 : Affectez le résultat de votre fonction pour `6` sommets à une variable `G` et affichez la variable `G`.

Question 7 : Écrivez une fonction `print_graph_matrix(g)` qui afficher la matrice du graphe sous la forme :

```
[0, 1, 1, 0, 0]
[1, 0, 1, 1, 0]
[1, 1, 0, 1, 1]
[0, 1, 1, 0, 0]
[0, 0, 1, 0, 0]
```

c'est-à-dire qui affiche une ligne après l'autre.

Question 8 : Affichez votre graphe `G` avec la fonction `print_graph_matrix`.

On fournit la fonction `draw_graph(matrix)` qui prend en argument une matrice d'adjacence et dessine le graphe non orienté associé.

Question 9 : Dessinez votre graphe en utilisant la fonction `draw_graph`. (pensez à exécuter votre code pour voir le dessin puis à le commenter comme expliqué au début de l'énoncé pour utiliser la correction automatique)

Question 10 : Affichez l'évaluation d'une expression booléenne qui vérifie si le sommet 0 et le sommet 3 sont voisins.

Question 11 : Affichez l'évaluation d'une expression booléenne qui vérifie si le sommet 2 et le sommet 4 sont voisins.

Correction

```
# Les deux lignes prochaines permettent d'ajouter de nouvelles fonctions dont vous aurez bes
from tp10_tools import draw_graph, symetrize
```

```
from random import randint


# Q1
def random_row(n):
    R = []
    for i in range(n):
        R.append(randint(0, 1))
    return R


# Q2
print(random_row(6))


# Q3
def random_oriented_graph(n):
    G = []
    for i in range(n):
        G.append(random_row(n))
    return G


# Q4
print(random_oriented_graph(3))


# Q5
def random_graph(n):
    G = random_oriented_graph(n)
    for i in range(len(G)):
        G[i][i] = 0
    G = symetrize(G)
    return G


# Q6
G = random_graph(6)
print(G)

# Q7
def print_graph_matrix(g):
    for line in g:
        print(line)
```

```
# Q8
print_graph_matrix(G)

# Q9
#draw_graph(G)

# Q10
print(G[0][3] == 1)

# Q11
print(G[2][4] == 1)
```

Ex. 2 — Modification de graphe

Ajout d'arête

On donne le graphe G = [[0,1,1,1,0], [1,0,1,0,0,], [1,1,0,1,1], [1,0,1,0,0],[0,0,1,0,0]]

Question 1 : Représentez graphiquement `G` avec la fonction donnée `draw_graph`.

Question 2 : Écrivez une fonction `add_edge(G,i,j)` qui permet d'ajouter l'arête (i,j) à un graphe G donné.

Question 3 : Ajoutez l'arête (3,4) au graphe `G`.

Question 4 : Représentez graphiquement le graphe après modification et vérifiez que l'arête a bien été ajoutée.

Suppression d'arête

Question 5 : Écrivez une fonction `delete_edge(G,v1,v2)` qui permet de supprimer l'arête (v1,v2) d'un graphe G donné.

Question 6 : Supprimez l'arête (3,4) du graphe `G`.

Question 7 : Représentez graphiquement le graphe après modification et vérifiez que l'arête a bien été supprimée.

Recherche des voisin

Question 8 : Écrivez une fonction `neighbours(G, node)` qui renvoie la liste des voisins du noeud `node` dans un graphe `G`.

Question 9 : Affichez les voisins du noeud 2 dans G.

Ajout d'un noeud

Question 10 : Écrivez une fonction `add_node(G, neighbours)` qui prend en argument un graphe et la liste d'adjacence du noeud que l'on veut créer et qui ajoute au graphe G un noeud ayant les voisins demandés. Par exemple, si on veut ajouter un noeud avec pour voisins 1,3 et 4, la liste neighbours vaudra [0,1,0,1,1].

Question 11 : Ajoutez un noeud à votre graphe avec comme seul voisin le noeud 2.

Question 12 : Représentez graphiquement le graphe après modification.

Suppression d'un noeud

Question 13 : Écrivez une fonction `delete_node(G, node)` qui prend en argument un graphe et le nom d'un noeud et qui supprime le noeud donné du graphe G. On rappelle qu'on peut utiliser del G[i] pour supprimer l'élément d'indice i de G.

Question 14 : Supprimez le noeud 5 de votre graphe.

Question 15 : Représentez graphiquement le graphe après modification.

Correction

```
# La ligne prochaine permet d'ajouter une nouvelle fonction dont vous aurez besoin après. Ne
from tp10_tools import draw_graph

G = [[0, 1, 1, 1, 0], [1, 0, 1, 0, 0, ], [1, 1, 0, 1, 1], [1, 0, 1, 0, 0], [0, 0, 1, 0, 0]]

# Q1
# draw_graph(G)


# Q2
def add_edge(G, v1, v2):
    if G[v1][v2] == 0:
        G[v1][v2] = 1
        G[v2][v1] = 1


# Q3
add_edge(G, 3, 4)

# Q4
```

```
# draw_graph(G)


# Q5
def delete_edge(G, v1, v2):
    if G[v1][v2] == 1:
        G[v1][v2] = 0
        G[v2][v1] = 0


# Q6
delete_edge(G, 3, 4)

# Q7
# draw_graph(G)


# Q8
def neighbours(G, node):
    l = []
    for i in range(len(G)):
        if G[node][i] == 1:
            l.append(i)
    return l


# Q9
print(neighbours(G, 2))


# Q10
def add_node(G, neighbours):
    for i in range(len(G)):
        G[i].append(neighbours[i])
    neighbours.append(0)
    G.append(neighbours)

# Q11
add_node(G, [0, 0, 1, 0, 0])

# Q12
# draw_graph(G)


# Q13
def delete_node(G, node):
```

```
    if node < len(G):
        del (G[node])
    for i in range(len(G)):
        del (G[i][node])


# Q14
delete_node(G, 5)

# Q15
# draw_graph(G)
```

Ex. 3 — Graphe eulérien

On donne le graphe G = [[0,1,1,1,0], [1,0,1,0,0,], [1,1,0,1,1], [1,0,1,0,0],[0,0,1,0,0]].

On définit le degré d'un noeud v du graphe G par le nombre d'arêtes ayant v comme extrémité.

Degré d'un noeud

Question 1 : Écrivez une fonction `node_degree(G, node)` qui renvoie le degré du noeud `node` dans le graphe G.

Question 2 : Affichez le degré du noeud 2 du graphe G.

Existence d'un cycle eulérien

Vous avez vu en Introduction à l'informatique qu'un graphe non orienté connexe admet un cycle qui traverse chaque arête exactement une fois (un cycle eulérien) si et seulement si chaque sommet est de degré pair.

Question 3 : Écrivez une fonction `eulerian_cycle(G)` qui renvoie `True` si le graphe `G` contient un cycle eulérien et `False` sinon.

Question 4 : On donne G = [[0,1,1,1], [1,0,1,1], [1,1,0,1], [1,1,1,0]] . Affichez si G contient un cycle eulérien ou non.

Question 5 : On donne G = [[0,1,1,1,1], [1,0,1,1,1], [1,1,0,1,1], [1,1,1,0,1],[1,1,1,1,0]]. Affichez si G contient un cycle eulérien ou non.

Correction

```
G = [[0,1,1,1,0], [1,0,1,0,0,], [1,1,0,1,1], [1,0,1,0,0],[0,0,1,0,0]]
```

```
# Q1
def node_degree(G, node):
    cpt = 0
    for edge in G[node]:
        if edge==1:
            cpt = cpt + 1
    return cpt


# Q2
print(node_degree(G,2))


# Q3
def eulerian_cycle(G):
    for node in range(len(G)):
        if node_degree(G, node) % 2 == 1:
            return False
    return True


# Q4
print(eulerian_cycle([[0,1,1,1], [1,0,1,1], [1,1,0,1], [1,1,1,0]]))


# Q5
print(eulerian_cycle([[0,1,1,1,1], [1,0,1,1,1], [1,1,0,1,1], [1,1,1,0,1],[1,1,1,1,0]]))
```