Algorithmique Appliquée en Python

Alexandre Meslé

24 novembre 2015

Table des matières

Chapitre 1

Notes de cours

1.1 Introduction

1.1.1 Hello World !

Le programme suivant permet d'afficher "Hello world" sur la console :

```
#!/usr/bin/python3

# Cette instruction affiche Hello world!
print ("Hello, world!")
```

Vous pouvez l'exécuter en ligne de commande avec l'instruction `./helloWorld.py`.

1.1.2 Quelques explications

Entête

La déclaration *#!/usr/bin/python3* indique quel est le programme qui va lire votre script Python pour l'exécuter. Ce programme s'appelle un **interpréteur**.

Sans interpréteur, il est impossible d'exécuter un programme Python. Mais en contrepartie, le même script Python peut être exécuté n'importe quelle plate-forme.

Corps du programme

```
# Cette instruction affiche Hello world!
print ("Hello, world!")
```

On place à la suite de l'entête les instructions que l'on souhaite exécuter. Dans le cas présent, un affichage.

```
print ("Hello, world!")
```

Pour afficher un message, on utilise **print**(`"message àafficher"`);. Les valeurs entre doubles quotes sont affichées telles quelles.

Commentaires

Un commentaire est une séquence de caractères ignorée par l'interpréteur, on s'en sert pour expliquer des portions de code. Quand vous coderez, ayez une pensée pour les pauvres programmeurs qui vont reprendre votre code après vous, le pauvre correcteur qui va essayer de comprendre votre pensée profonde, ou bien plus simplement à vous-même au bout de six mois, quand vous aurez complètement oublié ce que vous aviez dans la tête en codant.

On débute un commentaire avec le caractère #, le commentaire se termine alors à la fin de la ligne.

Il est aussi possible de délimiter un commentaire s'étendant sur plusieurs lignes par `"""`. Par exemple,

```
#!/usr/bin/python3
"""
Ceci est un commentaire :
L'instruction ci-dessous affiche ''Hello world!''
"""
print("Hello world!") # autre commentaire
```

1.2 Variables

Une variable est un emplacement de la mémoire dans lequel est stockée une valeur. Chaque variable porte une nom et c'est ce nom qui sert à identifier l'emplacement de la mémoire représenté par cette variable.

Contrairement à d'autres langages (en particulier les langages compilés), il n'est pas nécessaire en Python de déclarer les variables.

1.2.1 Affectation

Si on souhaite placer dans une variable `v` une valeur, on utilise l'opérateur =. Par exemple,

```
v = 5
```

Cet extrait de code crée déclare une variable `v`, puis lui affecte la valeur 5. Comme cette valeur est numérique, il ne sera plus possible par la suite d'affecter une valeur non numérique à `v`.

Il est possible d'appliquer des opérations arithmétiques pendant l'affectation :

— l'addition (+)
— la soustraction (−)
— la multiplication (∗)
— la division (/)
— la division entière (quotient : //, reste : %).
— l'exponentiation : (∗∗).

Par exemple,

```
v = 5
w = v + 1
z = v + w / 2
v = z % 3
v = v * 2
w = 5//2
z = z**2
```

1.2.2 Saisie

Traduisons l'instruction `Saisir` <`variable`> que nous avons vu en algorithmique. Pour récupérer la saisie d'un utilisateur et la placer dans une variable <`variable`>, on utilise l'instruction suivante :

```
<variable> = <type>(input())
```

Ne vous laissez pas impressionner par l'apparente complexité de cette instruction, elle suspend l'exécution du programme jusqu'à ce que l'utilisateur ait saisi une valeur et pressé la touche `return`. La valeur saisie est alors affectée à la variable <`variable`>.

<**type**> est le type de la variable à laquelle on affecte la valeur saisie.

1.2.3 Affichage

Traduisons maintenant l'instruction `Afficher` <`variable`> :

```
print(<variable>)
```

Cette instruction affiche la valeur contenue dans la variable <`variable`>. Nous avons étendu, en algorithmique, l'instruction `Afficher` en intercalant des valeurs de variables entre les messages affichés. Il est possible de faire de même en Python :

```
print("la valeur de la variable v est", v)
```

Les valeurs ou variables affichées sont ici séparés par des ,. Tout ce qui est délimité par des double quotes est affiché tel quel. Cette syntaxe s'étend à volonté :

```
print("les valeurs des variables x, y et z sont ", x,
      ", ", y, " et ", z)
```

1.2.4 Entiers

On utilise des variables de type **int** pour représenter des entiers. Il est nécessaire de le préciser en particulier lors d'une saisie. Par exemple, on saisit un **int** dans la variables `x` avec l'instruction suivante :

```
x = int(input())
```

1.2.5 Flottants

Les flottants servent à représenter les réels. Leur nom vient du fait qu'on les représente de façon scientifique : un nombre décimal (à virgule) muni d'un exposant (un décalage de la virgule). Le type servant à les représenter est **float**.

Notez bien le fait qu'un point flottant est un nombre avec lequel on peut "déplacer la virgule". Cela permet de représenter des valeurs très grandes ou très petites, mais au détriment de la précision. Nous examinerons dans les exercices les avantages et inconvénients des flottants.

Un littéral flottant s'écrit avec un point, par exemple l'approximation à 10^{-2} près de π s'écrit 3.14.

```
pi = 3.14
```

Il est aussi possible d'utiliser la notation scientifique, par exemple le décimal 1 000 s'écrit `1e3`, à savoir 1.10^3.

```
x = 1e3
```

1.2.6 Chaînes de caractères

Une chaîne de caractères, abrégé **str**, est une succession de caractères (aucun, un ou plusieurs). Les littéraux de ce type se délimitent par des quotes (doubles ou simples), et l'instruction de saisie s'écrit sans le <**type**>. Par exemple,

```
maChaine = input()
```

1.2.7 Caractères

Lorsqu'une chaîne ne contient qu'un symbole, elle contient en fait la code `ASCII` de ce symbole. On récupère ce code avec la fonction **ord**. Par exemple l'instruction suivante affecte à `code` le code du caractère `chaine`.

```
code = ord(chaine)
```

Réciproquement, on peut obtenir le caractère correspondant à un code `ASCII` avec la fonction **chr**. Par exemple l'instruction suivante affecte à `caractere` le symbole dont le code `ASCII` est 100.

```
caractere = chr(100)
```

1.2.8 Formes contractées

Le Python étant un langage de paresseux, tout à été fait pour que les programmeurs aient le moins de caractères possible à saisir. Je vous préviens : j'ai placé ce chapitre pour que soyez capable de décrypter la bouillie que pondent certains programmeurs, pas pour que vous les imitiez ! Alors vous allez me faire le plaisir de faire usage des formes contractées avec parcimonie, n'oubliez pas qu'il est très important que votre code soit **lisible**.

Toutes les affectations de la forme `variable = variable operateurBinaire expression` peuvent être contractées sous la forme `variable operateurBinaire= expression`. Par exemple,

avant	après
`a = a + b`	`a += b`
`a = a - b`	`a -= b`
`a = a * b`	`a *= b`
`a = a / b`	`a /= b`
`a = a % b`	`a %= b`
`a = a >> i`	`a >>= i`
`a = a << i`	`a <<= i`
`a = a & b`	`a &= b`
`a = a ^ b`	`a ^= b`
`a = a | b`	`a |= b`

Vous vous douterez que l'égalité ne peut pas être contractée...

1.3 Traitements conditionnels

On appelle traitement conditionnel une portion de code qui n'est pas exécutée systématiquement, c'est à dire des instructions dont l'exécution est conditionnée par le succès d'un test.

1.3.1 Si ... Alors

Principe

En algorithmique un traitement conditionnel se rédige de la sorte :

Si *condition* **alors**
| instructions
Fin

Si la condition est vérifiée, alors les instructions sont exécutées, sinon, elles ne sont pas exécutées. L'exécution de l'algorithme se poursuit alors en ignorant les instructions se trouvant entre le `alors` et le `finSi`. Un traitement conditionnel se code de la sorte :

```
if (<condition>):
        <instructions>
```

Notez bien qu'il y a **deux points** après la parenthèse du `if`.

Comparaisons

La formulation d'une condition se fait souvent à l'aide des opérateurs de comparaison. Les opérateurs de comparaison disponibles sont :

— == : égalité
— != : non-égalité
— <, <= : inférieur à, respectivement strict et large
— >, >= : supérieur à, respectivement strict et large

Par exemple, la condition `a == b` est vérifiée si et seulement si `a` et `b` ont la même valeur au moment où le test est évalué. Par exemple,

```
i = int(input("Saisissez une valeur"))
if (i == 0):
        print("Vous avez saisi une valeur nulle.")
print("Au revoir !")
```

Si au moment où le test `i == 0` est évalué, la valeur de `i` est bien 0, alors le test sera vérifié et l'instruction `print("Vous avez saisi une valeur nulle.")` sera bien exécutée. Si le test n'est pas vérifié, les instructions du bloc sous la portée du `if` sont ignorées.

Achtung !!!

Les indentations sont indispensables ! Les instructions sous la portée du `if` doivent être précédées d'une tabulation de plus que le `if`. Sinon des choses atroces se produiront !

Si ... Alors ... Sinon

Il existe une forme étendue de traitement conditionnel, on la note en algorithmique de la façon suivante :

Si *condition* **alors**
| instructions
Sinon
| autresinstructions
Fin

Les instructions délimitées par `alors` et `sinon` sont exécutées si le test est vérifié, et les instructions délimitées par `sinon` et `finSi` sont exécutées si le test n'est pas vérifié. On traduit le traitement conditionnel étendu de la sorte :

```
if (<condition>):
        <instructions1>
else:
        <instructions2>
```

Par exemple,

```
i = int(input("Saisissez une valeur"))
if (i == 0):
        print("Vous avez saisi une valeur nulle.")
else:
        print("La valeur que vous avez saisi n'est pas nulle.")
```

Notez la présence de l'opérateur de comparaison ==. **Si vous utilisez = pour comparer deux valeurs, ça ne compilera pas !**

Connecteurs logiques

On formule des conditions davantage élaborées en utilisant des connecteurs `et` et `ou`. La condition `A et B` est vérifiée si les deux conditions `A` et `B` sont vérifiées simultanément. La condition `A ou B` est vérifiée si au moins une des deux conditions `A` et `B` est vérifiée. Le `et` s'écrit **and** et le `ou` s'écrit **or**. Par exemple, voici un programme qui nous donne le signe de $i \times j$ sans les multiplier.

```
print("Saisissez deux valeurs numériques : ")
i = float(input())
j = float(input())
print("Le produit de ", i, " par ", j, " est ")
if ((i >= 0 and j >= 0) or (i <= 0 and j <= 0)):
        printLine("positif.")
else:
        printLine("négatif.")
```

1.3.2 Booléens

Une variable booléenne ne peut prendre que deux valeurs : *vrai* et *faux*. Le type booléen en Python est **bool** et une variable de ce type peut prendre soit la valeur `True`, soit la valeur `False`.

Utilisation dans des if

Lorsqu'une condition est évaluée, par exemple lors d'un test, cette condition prend à ce moment la valeur *vrai* si le test est vérifié, *faux* dans le cas contraire. Il est donc possible de placer une variable booléenne dans un if. Observons le test suivant :

```
b = bool(input("Saisissez un booléen : "))
if (b):
        printLine("b est vrai.")
else:
        printLine("b is faux.")
```

Si b contient la valeur `True`, alors le test est réussi, sinon le **else** est exécuté. On retiendra donc qu'il est possible de placer dans un **if** toute expression pouvant prendre les valeurs `True` ou `False`.

Tests et affectations

Un test peut être effectué en dehors d'un **if**, par exemple de la façon suivante :

```
x = (3>2)
```

On remarque que (3>2) est une condition. Pour décider quelle valeur doit être affectée à `x`, cette condition est évaluée. Comme dans l'exemple ci-dessus la condition est vérifiée, alors elle prend la valeur `True`, et cette valeur est affectée à `x`.

Connecteurs logiques binaires

Les connecteurs **or** et **and** peuvent s'appliquer à des valeurs (ou variables) booléennes. Observons l'exemple suivant :

```
x = (True and False) or (True)
```

Il s'agit de l'affectation à `x` de l'évaluation de la condition (`True` **and** `False`)**or** (`True`). Comme (`True` **and** `False`) a pour valeur `False`, la condition `False` **or** `True` est ensuite évaluée et prend la valeur `True`. Donc la valeur `True` est affectée à `x`.

Opérateur de négation

Parmi les connecteurs logiques se trouve **not**, dit opérateur de négation. La négation d'une expression est vraie si l'expression est fausse, fausse si l'expression est vraie. Par exemple,

```
x = not (3==2)
```

Comme 3 == 2 est faux, alors sa négation **not** (3 == 2) est vraie. Donc la valeur `True` est affectée à `x`.

1.4 Boucles

Nous souhaitons créer un programme qui nous affiche tous les nombres de 1 à 5, donc dont l'exécution serait la suivante :

```
1 2 3 4 5
```

Une façon particulièrement vilaine de procéder serait d'écrire 5 **print** successifs, avec la laideur des copier/coller que cela impliquerait. Nous allons étudier un moyen de coder ce type de programme avec un peu plus d'élégance.

1.4.1 Définitions et terminologie

Une boucle permet d'exécuter plusieurs fois de suite une même séquence d'instructions. Cette ensemble d'instructions s'appelle le **corps** de la boucle. Chaque exécution du corps d'une boucle s'appelle une **itération**, ou plus informellement un **passage** dans la boucle. Lorsque l'on s'apprête à exécuter la première itération, on dit que l'on **entre** dans la boucle, lorsque la dernière itération est terminée, on dit qu'on **sort** de la boucle. Il existe deux types de boucle :

— **while**
— **for**

Chacune de ces boucles a ses avantages et ses inconvénients. Nous les passerons en revue ultérieurement.

1.4.2 while

En Python, la boucle `tant que` se code de la façon suivante :

```
while(<condition>):
  <instructions>
```

Les instructions du corps de la boucle sont indentées. La condition est évaluée **avant** chaque passage dans la boucle, à chaque fois qu'elle est vérifiée, on exécute les instructions de la boucle. Un fois que la condition n'est plus vérifiée, l'exécution se poursuit après l'accolade fermante. Affichons par exemple tous les nombres de 1 à 5 dans l'ordre croissant,

```
i = 1
while (i <= 5):
    print(i)
    i += 1
```

Ce programme **initialise** i **à** 1 et tant que la valeur de i n'éxcède pas 5, cette valeur est affichée puis incrémentée. Les instructions se trouvant dans le corps de la boucle sont donc exécutées 5 fois de suite. La variable i s'appelle un **compteur**, on gère la boucle par incrémentations successives de i et on sort de la boucle une fois que i a atteint une certaine valeur. **L'initialisation du compteur est très importante** ! Si vous n'initialisez pas i explicitement, alors cette variable contiendra n'importe quelle valeur et votre programme ne se comportera pas du tout comme prévu. Notez bien par ailleurs qu'il y a deux points après la parenthèse.

1.4.3 for

On utilise une boucle **for** lorsque l'on connait en entrant dans la boucle combien d'itérations devront être faites. On évitera par exemple, de s'en servir pour contrôler une saisie.

Commençons par donner sa syntaxe :

```
for <compteur> in range (<premiere valeur>, <derniere valeur + 1>):
  <instructions>
```

Le `<compteur>` est déclaré dans l'entête de la boucle et est automatiquement initialisé à `<premiere valeur>`. Le compteur est ensuite automatiquement incrémenté entre les itérations jusqu'à ce qu'il atteigne la `<derniere valeur>`.

Par exemple, on réécrit l'affiche des 5 premiers entiers de la sorte :

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

Attention ! **Remarquez bien que la borne supérieure utilisée dans l'entête est** 6 !

1.5 Tableaux

Considérons un programme dont l'exécution donne :

```
Saisissez dix valeurs :
1 : 4
2 : 7
3 : 34
4 : 1
5 : 88
6 : 22
7 : 74
8 : 19
9 : 3
10 : 51
Saisissez une valeur :
22
22 est la 6-ème valeur saisie.
```

Comment programmer cela sans utiliser 10 variables pour stocker les dix premières valeurs saisies ?

1.5.1 Définitions

Une tableau est un regroupement de variables de même type, il est identifié par un nom. Chacune des variables du tableau est numérotée, ce numéro s'appelle un **indice**. Chaque variable du tableau est donc caractérisée par le nom du tableau et son indice.

Si par exemple, T est un tableau de 10 variables, alors chacune d'elles sera numérotée et il sera possible de la retrouver en utilisant simultanément le nom du tableau et l'indice de la variable. Les différentes variables de T porteront des numéros de 0 à 9, et nous appellerons chacune de ces variables un **élément** de T.

Une variable n'étant pas un tableau est appelée variable **scalaire**, un tableau par opposition à une variable scalaire est une variable **non scalaire**.

1.5.2 Déclaration

Comme les variables d'un tableau doivent être de même type, il convient de préciser ce type au moment de la délaration du tableau. De même, on précise lors de la déclaration du tableau le nombre de variables qu'il contient. La syntaxe est :

```
<nom> = array('<type>', range(0, <taille>))
```

Par exemple,

```
T = array('i', range(0, 4));
```

déclare un tableau `T` contenant 4 variables de type `int`.

Attention ! Vous devez placez au début de votre programme l'instruction

```
from array import array
```

1.5.3 Initialisation

Il est possible d'initialiser les éléments d'un tableau à la déclaration. On écrit pour cela entre crochets tous les éléments du tableau, on les dispose par ordre d'indice croissant en les séparant par des virgules. La syntaxe générale de la valeur d'initialisation est donc :

```
<nom> = array('<type>', [<valeur_0>, <valeur_1>, ..., <valeur_n-1>])
```

Par exemple, on crée un tableau contenant les 5 premiers nombres impairs de la sorte :

```
tab = array('i', [1, 3, 5, 7, 9])
```

1.5.4 Accès aux éléments

Les éléments d'un tableau à n éléments sont indicés de 0 à $n-1$. On note `T[i]` l'élément d'indice i du tableau `T`. Les cinq éléments du tableau de l'exemple ci-avant sont donc notés `T[0]`, `T[1]`, `T[2]`, `T[3]` et `T[4]`.

1.5.5 Exemple

Nous pouvons maintenant mettre en place le programme du début du cours. Il est nécessaire de stocker 10 valeurs de type entier, nous allons donc déclarer un tableau `e` de la sorte :

```
e = array('i', range(0, 10))
```

La déclaration ci-dessus est celle d'un tableau de 10 **int** appelé `e`. Il convient ensuite d'effectuer les saisies des 10 valeurs. On peut par exemple procéder de la sorte :

```
print ("Saisissez 10 valeurs")
e[0] = int(input("1 : "))
e[1] = int(input("2 : "))
e[2] = int(input("3 : "))
e[3] = int(input("4 : "))
e[4] = int(input("5 : "))
e[5] = int(input("6 : "))
e[6] = int(input("7 : "))
e[7] = int(input("8 : "))
e[8] = int(input("9 : "))
e[9] = int(input("10 : "))
```

Les divers copier/coller necéssaires pour rédiger un tel code sont d'une laideur à proscrire. Nous procéderons plus élégament en faisant une boucle :

```
for i in range(0, 10):
    e[i] = int(input(str(i + 1) + " : "))
```

Ce type de boucle s'appelle un **parcours** de tableau. En règle générale on utilise des boucles pour manier les tableaux, celles-ci permettent d'effectuer un traitement sur chaque élément d'un tableau. Ensuite, il faut saisir une valeur à rechercher dans le tableau :

```
t = int(input("Saisissez une valeur : "))
```

Nous allons maintenant rechercher la valeur `t` dans le tableau `e`. Considérons pour ce faire la boucle suivante :

```
i = 0
while e[i] != t:
        i = i + 1
```

Cette boucle parcourt le tableau jusqu'à trouver un élément de `e` qui ait la même valeur que `t`. Le problème qui pourrait se poser est que si `t` ne se trouve pas dans le tableau `e`, alors la boucle pourrait ne pas s'arrêter. Si `i` prend des valeurs strictement plus grandes que 9, alors il se produira ce que l'on appelle un **débordement d'indice**. Vous devez toujours veiller à ce qu'il ne se produise pas de débordement d'indice ! Nous allons donc faire en sorte que la boucle s'arrête si `i` prend des valeurs strictement supérieures à 9.

```
i = 0
while i < 10 and e[i] != t:
        i = i + 1
```

Il existe donc deux façons de sortir de la boucle :

— En cas de débordement d'indice, la condition `i < 10` ne sera pas vérifiée. Une fois sorti de la boucle, `i` aura la valeur 10.

— Dans le cas où `t` se trouve dans le tableau à l'indice `i`, alors la condition `e[i] != t` ne sera pas vérifiée et on sortira de la boucle. Un fois sorti de la boucle, `i` aura comme valeur l'indice de l'élément de `e` qui est égal à `t`, donc une valeur comprise entre 0 et 9.

On identifie donc la façon dont on est sorti de la boucle en testant la valeur de `i` :

```
if i == 10:
    print (str(t) + " ne fait pas partie des 10 valeurs saisies.")
else:
    print (str(t) + " est la " + str(i + 1) + "-ème valeur saisie.")
```

Si (`i == 10`), alors nous sommes sorti de la boucle parce que l'élément saisi par l'utilisateur ne trouve pas dans le tableau. Dans le cas contraire, `t` est la `i`+1-ème valeur saisie par l'utilisateur. On additionne 1 à l'indice parce que l'utilisateur ne sait pas que dans le tableau les éléments sont indicés à partir de 0. Récapitulons :

```
# -*- coding: utf-8 -*-

from array import array

taille = 10
print ("Saisissez " + str(taille) + " valeurs")
e = array('i', range(0, taille))
for i in range(0, len(e)):
    e[i] = int(input(str(i + 1) + " : "))
t = int(input("Saisissez une valeur : "))
i = 0;
while i < len(e) and t != e[i]:
    i += 1
if i == len(e):
    print (str(t) + " ne fait pas partie des " + str(len(e)) + " valeurs saisies.")
else:
    print (str(t) + " est la " + str(i + 1) + "-ème valeur saisie.")
```

1.5.6 La boucle for simplifiée

Il est possible de parcourir un tableau de façon séquentielle et en lecture seule grâce à l'instruction **for**.

```
for <variable> in <tableau>:
                <instructions>
```

Cette instruction affecte à <`variable`> pour chaque itération de la boucle un élément de <`tableau`>. On remarque au passage que le <**type**> doit être celui des éléments de <`tableau`>. Par exemple, le programme suivant affiche les quatre premières lettres de l'alphabet :

```
t = array('u', ['a', 'b', 'c', 'd'])
for c in t:
        print(c);
```

1.6 Sous-programmes

1.6.1 Les procédures

Une **procédure** est un **ensemble d'instructions** portant un **nom**. Pour définir une procédure, on utilise la syntaxe :

```
def nomprocedure():
        instructions
```

Une procédure est une nouvelle instruction, il suffit pour l'exécuter d'utiliser son nom. Par exemple, pour **exécuter** (on dit aussi **appeler** ou **invoquer**) une procédure appelée *pr*, il suffit d'écrire *pr*();. Les deux programmes suivants font la même chose. Le premier est écrit sans procédure :

Vous remarquez les instructions sous la portée de la procédure sont décalées d'une tabulation.

```
print ("Bonjour !")
```

Et dans le deuxième, le **print** est placé dans la procédure `afficheBonjour` et cette procédure est appelée depuis le script principal.

```
def afficheBonjour():
        print ("Bonjour !")

afficheBonjour()
```

Vous pouvez définir autant de procédures que vous le voulez et vous pouvez appeler des procédures depuis des procédures :

```
def afficheBonjour():
    print("Bonjour,")

def afficheUn():
    print("1")

def afficheDeux():
    print("2")

def afficheUnEtDeux():
    afficheUn()
    afficheDeux()

def afficheAuRevoir():
    print("Au revoir.")

afficheBonjour()
afficheUnEtDeux()
afficheAuRevoir()
```

Ce programme affiche :

```
Bonjour,
1
2
Au revoir.
```

Regardez bien le programme suivant et essayez de déterminer ce qu'il affiche.

```
def procedure1():
    print("debut procedure 1")
    print("fin procedure 1")
```

```
def procedure2():
    print("debut procedure 2")
    procedure1()
    print("fin procedure 2")

def procedure3():
    print("debut procedure 3")
    procedure1()
    procedure2()
    print("fin procedure 3")

print("debut main")
procedure2()
procedure3()
print("fin main")
```

La réponse est

```
debut main
debut procedure 2
debut procedure 1
fin procedure 1
fin procedure 2
debut procedure 3
debut procedure 1
fin procedure 1
debut procedure 2
debut procedure 1
fin procedure 1
fin procedure 2
fin procedure 3
fin main
```

1.6.2 Variables locales

Il est possible de déclarer des variables à l'intérieur d'une procédure.

```
def nomprocedure():
        i=0
        print (i)
```

Attention, ces variables ne sont accessibles que dans le corps de la procédure, cela signifie qu'elles naissent au moment de leur déclaration et qu'elles sont détruites une fois la dernière instruction de la procédure exécutée. C'est pour cela qu'on les apelle des **variables locales**. Une variable locale n'est **visible** qu'entre sa déclaration et la fin de la procédure. Par exemple, ce code est illégal :

```
def maProcedure():
        a = b

b = 'k'
print(a)
print(b)
```

En effet, la variable b est déclarée dans la procédure `Main`, et n'est donc visible que dans cette même procédure. Son utilisation dans `maProcedure` est donc impossible. De même, la variable a est déclarée dans `maProcedure`, elle n'est pas visible dans le `Main`. Voici un exemple d'utilisation de variables locales :

```
def unADix():
    for i in range(1, 11):
```

```
        print(i)

unADix()
```

1.6.3 Passage de paramètres

Il est possible que la valeur d'une variable locale d'une procédure ne soit connue qu'au moment de l'appel de la procédure. Considérons le programme suivant :

```
i = int(input("Veuillez saisir un entier : "))
# Appel d'une procedure affichant la valeur de i.
# ...
```

Comment définir et invoquer une procédure `afficheInt` permettant d'afficher cet entier saisi par l'utilisateur ? Vous conviendrez que la procédure suivante ne passera pas la compilation

```
def afficheInt():
        print(i)
```

En effet, la variable i est déclarée dans le `Main`, elle n'est donc pas visible dans `afficheInt`. Pour y remédier, on définit `afficheInt` de la sorte :

```
def afficheInt(i):
        print(i)
```

i est alors appelé un **paramètre**, il s'agit d'une variable dont la valeur sera précisée lors de l'appel de la procédure. On peut aussi considérer que i est une valeur inconnue, et qu'elle est **initialisée** lors de l'invocation de la procédure. Pour initialiser la valeur d'un paramètre, on place cette valeur entre les parenthèses lors de l'appel de la procédure, par exemple : `afficheInt`(4) lance l'exécution de la procédure `afficheInt` en initialisant la valeur de i à 4. On dit aussi que l'on **passe en paramètre** la valeur 4. La version correcte de notre programme est :

```
def afficheInt(i):
        print(i)

i = int(input("Veuillez saisir un entier : "))
afficheInt(i)
```

Attention, notez bien que le i de `afficheInt` et le i du `Main` sont **deux variables différentes**, la seule chose qui les lie vient du fait que l'instruction `afficheInt`(`i`) initialise le i de `afficheInt` à la valeur du i du `Main`. Il serait tout à fait possible d'écrire :

```
def afficheInt(j):
        print(j)

i = int(input("Veuillez saisir un entier : "))
afficheInt(i)
```

Dans cette nouvelle version, l'instruction `afficheInt`(`i`) initialise le j de `afficheInt` à la valeur du i du `Main`.

Il est possible de passer plusieurs valeurs en paramètre. Par exemple, la procédure suivante affiche la somme des deux valeurs passées en paramètre :

```
def afficheSomme(a, b):
        print(a + b)
```

L'invocation d'une telle procédure se fait en initialisant les paramètres dans le même ordre et en séparant les valeurs par des virgules, par exemple `afficheSomme`(3, 5) invoque `afficheSomme` en initialisant a à 3 et b à 5. Vous devez initialiser tous les paramètres et vous devez placer les valeurs dans l'ordre. Récapitulons :

```
def afficheSomme(a, b):
        print(a + b)

i = int(input("Veuillez saisir deux entiers :\na = "))
j = int(input("b = "))
print("a + b = ")
afficheSomme(i, j)
```

La procédure ci-avant s'exécute de la façon suivante :

```
Veuillez saisir deux entiers :
a = 3
b = 5
a + b = 8
```

Lors de l'appel `afficheInt(r)` de la procédure `public static void afficheInt(int i)`, r est le **paramètre effectif** et i le **paramètre formel**. Notez bien que i et r sont deux variables distinctes. Par exemple, qu'affiche le programme suivant ?

```
def incr(v):
        v += 1

i = 6
incr(i)
print(i)
```

La variable v est initialisée à la valeur de i, mais i et v sont deux variables différentes. Modifier l'une n'affecte pas l'autre.

1.6.4 Les fonctions

Le principe

Nous avons vu qu'un sous-programme appelant peut communiquer des valeurs au sous-programme appelé. Mais est-il possible pour un sous-programme appelé de communiquer une valeur au sous-programme appelant ? La réponse est oui. Une **fonction** est un sous-programme qui communique une valeur au sous-programme appelant. Cette valeur s'appelle **valeur de retour**, ou **valeur retournée**.

Invocation

La syntaxe pour appeler une fonction est :

```
v = nomfonction(parametres)
```

L'instruction ci-dessus place dans la variable v la valeur retournée par la fonction `nomfonction` quand lui passe les paramètres `parametres`. Nous allons plus loin dans ce cours définir une fonction `carre` qui retourne le carré de valeur qui lui est passée en paramètre, alors l'instruction

```
v = carre(2)
```

placera dans v le carré de 2, à savoir 4. On définira aussi une fonction `somme` qui retourne la somme de ses paramètres, on placera donc la valeur $2 + 3$ dans v avec l'instruction

```
v = somme(2, 3)
```

Définition

L'instruction servant à retourner une valeur est **return**. Cette instruction interrompt l'exécution de la fonction et retourne la valeur placée immédiatement après. Par exemple, la fonction suivante retourne toujours la valeur 1.

```
def un():
        return 1
```

Lorsque l'on invoque cette fonction, par exemple

```
v = un()
```

La valeur 1, qui est retournée par `un` est affectée à v. On définit une fonction qui retourne le successeur de son paramètre :

```
def successeur(i):
        return i + 1
```

Cette fonction, si on lui passe la valeur 5 en paramètre, retourne 6. Par exemple, l'instruction

```
v = successeur(5)
```

affecte à v la valeur 6. Construisons Maintenant nos deux fonctions :

```
def carre(i):
        return i * i

def somme(a, b):
        return a + b
```

1.6.5 Passages de paramètre par référence

En Python, lorsque vous invoquez une fonction, toutes les valeurs des paramètres effectifs sont recopiés dans les paramètres formels. On dit dans ce cas que le passage de paramètre se fait **par valeur**. Vous ne pouvez donc, *a priori*, communiquer qu'une seule valeur au programme appelant. Par conséquent **seule la valeur de retour vous permettra de communiquer une valeur au programme appelant**.

Lorsque vous passez un tableau en paramètre, la valeur qui est recopiée dans le paramètre formel est l'adresse de ce tableau (l'adresse est une valeur scalaire). Par conséquent toute modification effectuée sur les éléments d'un tableau dont l'adresse est passée en paramètre par valeur sera repercutée sur le paramètre effectif (i.e. le tableau d'origine). Lorsque les modifications faites sur un paramètre formel dans un sous-programme sont repércutées sur le paramètre effectif, on a alors un **passage de paramètre par référence**.

```
from array import array

def initTab(tableau):
    for i in range (0, len(tableau)):
        tableau[i] = i+1

def copieTab(original):
    copie = array('i', original)
    return copie

tableau = array('i', range(0, 20))
initTab(tableau)
autreTableau = copieTab(tableau)
for x in autreTableau:
    print(x)
```

Nous retiendrons donc les règles d'or suivantes :

— Les variables scalaires se passent en paramètre par valeur

— Les variables non scalaires se passent en paramètre par référence

Chapitre 2

Exercices

2.1 Variables

2.1.1 Saisie et affichage

Exercice 1 Saisie d'une chaîne

Écrire un programme demandant à l'utilisateur de saisir son nom, puis affichant le nom saisi.

Exercice 2 Saisie d'un entier

Ecrire un programme demandant à l'utilisateur de saisir une valeur numérique entière puis affichant cette valeur.

Exercice 3 Permutation de 2 variables

Saisir deux variables et les permuter avant de les afficher.

Exercice 4 Permutation de 4 valeurs

Ecrire un programme demandant à l'utilisateur de saisir 4 valeurs A, B, C, D et qui permute les variables de la façon suivante :

noms des variables	A	B	C	D
valeurs avant la permutation	1	2	3	4
valeurs après la permutation	3	4	1	2

Exercice 5 Permutation de 5 valeurs

On considère la permutation qui modifie cinq valeurs de la façon suivante :

noms des variables	A	B	C	D	E
valeurs avant la permutation	1	2	3	4	5
valeurs après la permutation	4	3	5	1	2

Ecrire un programme demandant à l'utilisateur de saisir 5 valeurs que vous placerez dans des variables appelées A, B, C, D et E. Vous les permuterez ensuite de la façon décrite ci-dessus.

Exercice 6 Permutation ultime

Même exercice avec :

noms des variables	A	B	C	D	E	F
valeurs avant la permutation	1	2	3	4	5	6
valeurs après la permutation	3	4	5	1	6	2

2.1.2 Entiers

Exercice 7 Opérations sur les entiers

Saisir deux variables entières, afficher leur somme et leur quotient.

2.1.3 Flottants

Exercice 8 Saisie et affichage

Saisir une variable de type `float`, afficher sa valeur.

Exercice 9 Moyenne arithmétique

Saisir 3 valeurs, afficher leur moyenne.

Exercice 10 Surface du rectangle

Demander à l'utilisateur de saisir les longueurs et largeurs d'un rectangle, afficher sa surface.

2.1.4 Caractères

Exercice 11 Prise en main

Affectez le caractère `'a'` à une variable de type `char`, affichez ce caractère ainsi que son code `ASCII`.

Exercice 12 Casse

Écrivez un programme qui saisit un caractère miniscule et qui l'affiche en majuscule.

2.1.5 Conversions

Exercice 13 Successeur

Écrivez un programme qui saisit un caractère et qui affiche son successeur dans la table des codes `ASCII`.

Exercice 14 Codes `UNICODE`

Quels sont les codes `UNICODE` des caractères `'0'`, `'1'`, ..., `'9'`

Exercice 15 Moyennes arithmétique et géométrique

Demander à l'utilisateur de saisir deux valeurs a et b et type **float**. Afficher ensuite la différence entre la moyenne arithmétique $\frac{(a+b)}{2}$ et la moyenne géométrique $\sqrt{ab}$. Vous utiliserez l'instruction `a**b`, qui donne la valeur a^b.

Exercice 16 Cartons et camions

Nous souhaitons ranger des cartons pesant chacun k kilos dans un camion pouvant transporter M kilos de marchandises. Ecrire un programme C demandant à l'utilisateur de saisir M et k, que vous représenterez avec des nombres flottants, et affichant le nombre (entier) de cartons qu'il est possible de placer dans le camion.

2.1.6 Morceaux choisis (difficiles)

Exercice 17 Pièces de monnaie

Nous disposons d'un nombre illimité de pièces de 0.5, 0.2, 0.1, 0.05, 0.02 et 0.01 euros. Nous souhaitons, étant donné une somme S, savoir avec quelles pièces la payer de sorte que le nombre de pièces utilisée soit minimal. Par exemple, la somme de 0.96 euros se paie avec une pièce de 0.5 euros, deux pièces de 0.2 euros, une pièce de 0.05 euros et une pièce de 0.01 euros.

1. Le fait que la solution donnée pour l'exemple est minimal est justifié par une idée plutôt intuitive. Expliquez ce principe sans excès de formalisme.
2. Ecrire un programme demandant à l'utilisateur de saisir une valeur comprise entre 0 et 0.99. Ensuite, affichez le détail des pièces à utiliser pour constituer la somme saisie avec un nombre minimal de pièces.

Exercice 18 Associativité de l'addition flottante

L'ensemble des flottants n'est pas associatif, cela signifie qu'il existe trois flottants a, b et c, tels que $(a+b)+c \neq a+(b+c)$. Trouvez de tels flottants et vérifiez-le dans un programme.

Exercice 19 Permutation sans variable temporaire

Permutez deux variables `a` et `b` sans utiliser de variable temporaire.

2.2 Conditions

2.2.1 Prise en main

Exercice 1 Majorité

Saisir l'âge de l'utilisateur et lui dire s'il est majeur ou s'il est mineur.

Exercice 2 Valeur absolue

Saisir une valeur, afficher sa valeur absolue.

Exercice 3 Admissions

Saisir une note, afficher "ajourné" si la note est inférieure à 8, "rattrapage" entre 8 et 10, "admis" dessus de 10.

2.2.2 Morceaux choisis

Exercice 4 Assurances

Une compagnie d'assurance effectue des remboursements sur lesquels est ponctionnée une franchise correspondant à 10% du montant à rembourser. Cependant, cette franchise ne doit pas excéder 4000 euros. Demander à l'utilisateur de saisir le montant des dommages, afficher ensuite le montant qui sera remboursé ainsi que la franchise.

Exercice 5 Valeurs distinctes parmi 2

Afficher sur deux valeurs saisies le nombre de valeurs distinctes.

Exercice 6 Plus petite valeur parmi 3

Afficher sur trois valeurs saisies la plus petite.

Exercice 7 Recherche de doublons

Ecrire un algorithme qui demande à l'utilisateur de saisir trois valeurs et qui lui dit s'il s'y trouve un doublon.

Exercice 8 Valeurs distinctes parmi 3

Afficher sur trois valeurs saisies le nombre de valeurs distinctes.

Exercice 9 $ax+b=0$

Saisir les coefficients a et b et afficher la solution de l'équation $ax+b=0$.

Exercice 10 $ax^2 + bx + c = 0$

Saisir les coefficients a, b et c, afficher la solution de l'équation $ax^2 + bx + c = 0$.

Exercice 11 Calculatrice

Écrire un programme demandant à l'utilisateur de saisir
— deux valeurs a et b, de type *int* ;
— un opérateur *op* de type *char*, vérifiez qu'il s'agit del'une des valeurs suivantes : $+, -, *, /$.
Puis affichez le résultat de l'opération a *op* b.

2.3 Boucles

2.3.1 Compréhension

Exercice 1

Qu'affiche le programme suivant ?

```
a = 1
b = 0
n = 5
while (a <= n):
    b += a
    a += 1
print(a, ", ", b)
```

Exercice 2

Qu'affiche le programme suivant ?

```
c = 0
m = 3
n = 4
for a in range(0, m):
    d = 0
    for b in range(0, n):
        d += b
        c += d
print(a, ", ", b, ", ", c, ", ", d, ".")
```

Exercice 3

Qu'affiche le programme suivant ?

```
a = 1
b = 2
c = a/b
if (a==b):
    d = 3
else:
    d= 4
print(c, ", ", d, ".")
b += 1
a = b
b %= 3
print(a, ", ", b, ".")
b = 1
```

```
for a in range(0, 10):
    c = ++b;
print(a, ", ", b, ", ", c, ", ", d, ".")
```

2.3.2 Utilisation de toutes les boucles

Les exercices suivants seront rédigés avec les trois types de boucle : tant que, répéter jusqu'à et pour.

Exercice 4 Compte à rebours

Écrire un programme demandant à l'utilisateur de saisir une valeur numérique positive n et affichant toutes les valeurs $n, n-1, \ldots, 2, 1, 0$.

Exercice 5 Factorielle

Ecrire un programme calculant la factorielle (factorielle $n = n! = 1 \times 2 \times \ldots \times n$ et $0! = 1$) d'un nombre saisi par l'utilisateur.

2.3.3 Choix de la boucle la plus appropriée

Pour les exercices suivants, vous choisirez la boucle la plus simple et la plus lisible.

Exercice 6 Table de multiplication

Écrire un programme affichant la table de multiplication d'un nombre saisi par l'utilisateur.

Exercice 7 Tables de multiplications

Écrire un programme affichant les tables de multiplications des nombres de 1 à 10 dans un tableau à deux entrées.

Exercice 8 Puissance

Écrire un programme demandant à l'utilisateur de saisir deux valeurs numériques b et n (vérifier que n est positif) et affichant la valeur b^n.

Exercice 9 Joli carré

Écrire un programme qui saisit une valeur n et qui affiche le carré suivant ($n = 5$ dans l'exemple) :

```
n = 5
X  X  X  X  X

X  X  X  X  X

X  X  X  X  X

X  X  X  X  X

X  X  X  X  X
```

2.3.4 Morceaux choisis

Exercice 10 Approximation de 2 par une série

On approche le nombre 2 à l'aide de la série $\sum_{i=0}^{+\infty} \frac{1}{2^i}$. Effectuer cette approximation en calculant un grand nombre de termes de cette série. L'approximation est-elle de bonne qualité ?

Exercice 11 Approximation de e par une série

Mêmes questions qu'à l'exercice précédent en e à l'aide de la série $\sum_{i=0}^{+\infty} \frac{1}{i!}$.

2.4 Tableaux

2.4.1 Exercices de compréhension

Qu'affichent les programmes suivants ?

Exercice 1

```
from array import array

c = array('u', ['a', 'b', 'c', 'd'])
c[0] = 'a';
c[3] = 'J';
c[2] = 'k';
c[1] = 'R';
for z in c:
        print(z)
for k in range(0, 4):
        c[k] = chr(ord(c[k]) + 1)
for z in c:
        print(z)
```

Exercice 2

```
from array import array

k = array('i', range(0, 10))
k[0] = 1;
for i in range(1, 10):
    k[i] = 0
for j in range (1, 4):
    for i in range(1, 10):
        k[i] += k[i - 1];
for i in k:
        print(i);
```

Exercice 3

```
from array import array

k = array('i', range(0, 10))
k[0] = 1
k[1] = 1
for i in range(2, 10):
    k[i] = 0;
for j in range (1, 4):
    for i in range (1, 10):
        k[i] += k[i - 1];
for p in k:
    print(p);
```

2.4.2 Prise en main

Exercice 4 - Initialisation et affichage

Ecrire un programme plaçant dans un tableau `T` les valeurs $1, 2, \ldots, 10$, puis affichant ce tableau. Vous initialiserez le tableau à la déclaration.

Exercice 5 - Initialisation avec une boucle

Même exercice en initialisant le tableau avec une boucle, vous placerez dans le tableau les 10 premiers nombres pairs.

Exercice 6 - Somme

Placez dans un tableau les 10 premiers nombres impairs, affichez la somme de ces nombres.

2.4.3 Indices

Exercice 7 - Permutation circulaire

Placez dans un deuxième tableau la permutation circulaire vers la droite des éléments d'un premier tableau.

Exercice 8 - Permutation circulaire sans deuxième tableau

Même exercice mais sans utiliser de deuxième tableau.

Exercice 9 - Miroir

Inversez l'ordre des éléments d'un tableau sans utiliser de deuxième tableau.

2.4.4 Recherche séquentielle

Exercice 10 - Modification du tableau

Créez un tableau `t` à 20 éléments. Placez dans `t[i]` le reste modulo 17 de i^2.

Exercice 11 - Min/max

Affichez les valeurs du plus petit et du plus grand élément de `T`.

Exercice 12 - Recherche séquentielle

Demandez à l'utilisateur de saisir une valeur `x` et donnez-lui la liste des indices `i` tels que `T[i]` a la valeur `x`.

Exercice 13 - Recherche séquentielle avec stockage des indices

Même exercice que précédemment, mais vous en affichant `La valeur ... se trouve aux indices suivants : ...` si `x` se trouve dans `T`, et `La valeur ... n'a pas été trouvée` si `x` ne se trouve pas dans `T`. Vous utiliserez un tableau `Q` dans lequel vous stockerez les indices auxquels `x` aura été trouvé dans `T`.

2.4.5 Morceaux choisis

Exercice 14 - Pièces de monnaie

Reprenez l'exercice sur les pièces de monnaie en utilisant deux tableaux, un pour stocker les valeurs des pièces dans l'ordre décroissant, l'autre pour stocker le nombre de chaque pièce.

Exercice 15 - Impôt sur le revenu

Refaites le programme de calcul de l'impôt sur le revenu en utilisant des tableaux.

2.5 Sous-programme

2.5.1 Géométrie

Ecrire un programme demandant à l'utilisateur de saisir une valeur numérique positive n et affichant un carré, une croix et un losange, tous de coté n. Par exemple, si $n = 10$, l'exécution donne

```
Saisissez la taille des figures
10
* * * * * * * * * *
*                 *
*                 *
*                 *
*                 *
*                 *
*                 *
*                 *
*                 *
* * * * * * * * * *
                  *
                *   *
              *       *
            *           *
          *               *
        *                   *
      *                       *
    *                           *
  *                               *
*                                   *
  *                               *
    *                           *
      *                       *
        *                   *
          *               *
            *           *
              *       *
                *   *
                  *
*                                   *
  *                               *
    *                           *
      *                       *
        *                   *
          *               *
            *           *
              *       *
                *   *
                  *
                *   *
              *       *
            *           *
          *               *
        *                   *
      *                       *
    *                           *
```

```
  *                               *
*                                   *
```

Vous définirez des sous-programmes de quelques lignes et au plus deux niveaux d'imbrication. Vous ferez attention à ne jamais écrire deux fois les mêmes instructions. Pour ce faire, complétez le code source suivant :

```
"""
Affiche le caractère c
"""

def afficheCaractere(c):
    return

"""
Affiche n fois le caractère c, ne revient pas à la ligne
après le dernier caractère.
"""

def ligneSansReturn(n, c):
    return
"""
Affiche n fois le caractère c, revient à la ligne après
le dernier caractère.
"""

def ligneAvecReturn(n, c):
    return

"""
Affiche n espaces.
"""

def espaces(n):
    return

"""
Affiche le caractère c a la colonne i,
ne revient pas à la ligne après.
"""

def unCaractereSansReturn(i, c):
    return

"""
Affiche le caractère c à la colonne i,
revient à la ligne après.
"""

def unCaractereAvecReturn(i, c):
    return

"""
Affiche le caractère c aux colonnes i et j,
revient à la ligne après.
"""

def deuxCaracteres(i, j, c):
    return

"""
```

```
Affiche un carré de côté n.
"""

def carre(n):
    return

"""
Affiche un chapeau dont la pointe - non affichée - est
sur la colonne centre, avec les caractères c.
"""

def chapeau(centre, c):
    return

"""
Affiche un chapeau à l'envers avec des caractères c,
la pointe - non affichée - est à la colonne centre.
"""

def chapeauInverse(centre, c):
    return

"""
Affiche un losange de côté n.
"""

def losange(n):
    return

"""
Affiche une croix de côté n.
"""

def croix(n):
    return

taille = int(input("Saisissez la taille des figures : "))
carre(taille)
losange(taille)
croix(taille)
```

2.5.2 Arithmétique

Exercice 1 - chiffres et nombres

1. Écrire la fonction `unites(n)` retournant le chiffre des unités du nombre n.
2. Écrire la fonction `dizaines(n)` retournant le chiffre des dizaines du nombre n.
3. Écrire la fonction `extrait(n, p)` retourant le p-ème chiffre de représentation décimale de n en partant des unités.
4. Écrire la fonction `nbChiffres(n)` retournant le nombre de chiffres que comporte la représentation décimale de n.
5. Écrire la fonction `sommeChiffres(n)` retournant la somme des chiffres de n.

Exercice 2 - Nombres amis

Soient a et b deux entiers strictement positifs. a est un diviseur strict de b si a divise b et $a \neq b$. Par exemple, 3 est un diviseur strict de 6. Mais 6 n'est pas un diviseur strict 6. a et b sont des nombres amis si la somme des diviseurs stricts de a est b et si la somme des diviseurs de b est a. Le plus petit couple de nombres amis connu est 220 et 284.

1. Écrire une fonction `sommeDiviseursStricts(n)`, elle doit renvoyer la somme des diviseurs stricts de n.

2. Écrire une fonction `sontAmis(a, b)`, elle doit renvoyer $True$ si a et b sont amis, $False$ sinon.

Exercice 3 - Nombres parfaits

Un nombre parfait est un nombre égal à la somme de ses diviseurs stricts. Par exemple, 6 a pour diviseurs stricts 1, 2 et 3, comme $1 + 2 + 3 = 6$, alors 6 est parfait.

1. Est-ce que 18 est parfait ?
2. Est-ce que 28 est parfait ?
3. Que dire d'un nombre ami avec lui-même ?
4. Écrire la fonction `estParfait(n)`, elle doit retourner $True$ ssi n est un nombre parfait.

Exercice 4 - Nombres de Kaprekar

Un nombre n est un nombre de Kaprekar en base 10, si la représentation décimale de n^2 peut être séparée en une partie gauche u et une partie droite v tel que $u + v = n$. $45^2 = 2025$, comme $20 + 25 = 45$, 45 est aussi un nombre de Kaprekar. $4879^2 = 23804641$, comme $238 + 04641 = 4879$ (le 0 de 046641 est inutile, je l'ai juste placé pour éviter toute confusion), alors 4879 est encore un nombre de Kaprekar.

1. Est-ce que 9 est un nombre de Kaprekar ?
2. Écrire la fonction `sommeParties(n, p)` qui découpe n est deux nombres dont le deuxième comporte p chiffres, et qui retourne leur somme. Par exemple,

$$sommeParties(12540, 2) = 125 + 40 = 165$$

3. Écrire la fonction `estKaprekar(int n)`

2.5.3 Passage de tableaux en paramètre

Exercice 5 - Somme

Écrire une fonction `somme(t)` retournant la somme des éléments de t.

Exercice 6 - Minimum

Écrire une fonction `min(t)` retournant la valeur du plus petit élément de t.

Exercice 7 - Recherche

Écrire une fonction `existe(t, k)` retournant `True` si et seulement si k est un des éléments de t.

Exercice 8 - Somme des éléments pairs

Écrivez le corps de la fonction `sommePairs(t)`, $sommePairs(t)$ retourne la somme des éléments pairs de t. N'oubliez pas que $a\%b$ est le reste de la division entière de a par b.

Exercice 9 - Vérification

Écrivez le corps de la fonction `estTrie(t)`, $estTrie(t)$ retourne vrai si et seulement si les éléments de t sont triés dans l'ordre croissant.

Exercice 10 - Permutation circulaire

Écrire une fonction `permutation(t)` effectuant une permutation circulaire vers la droite des éléments de t.

Exercice 11 - Miroir

Écrire une fonction `miroir(t` inversant l'ordre des éléments de t.