Types de données

C. Charignon

Table des matières

Première partie
Cours

1 Introduction, et types simples

On a besoin d'enregistrer différents types de données sur un ordinateur. Quelques exemples, vus par un utilisateur quelconque : nombres, textes, images, vidéos, musiques...

Dans tous les cas, il n'y aura qu'une suite de 0 et de 1 dans le disque dur. Simplement, le logiciel utilisé interprétera différemment cette suite de bits selon qu'il considère qu'il s'agit d'un nombre entier, d'un flottant, d'un texte, d'une vidéo, d'une image, etc. À notre niveau, nous n'aurons pas à nous préoccuper de ceci : ce sera automatiquement géré par Python. Il faut juste avoir à l'esprit que certains types de donnée prendront plus de place en mémoire, et seront plus long à manipuler.

Chaque langage de programmation propose un certain nombre de types de données, et permet éventuellement au programmeur de créer ses propres types.

Dans ce chapitre, nous verrons comment manipuler quelques types simples, disponibles dans la plupart des langages.

En Python, pour connaître le type de la donnée enregistrée dans une variable, on peut utiliser `type(maVariable)`.

Les informations à avoir à l'esprit concernant un type sont :

- la place mémoire occupée par une donnée ;
- les opérations permises ;
- la vitesses de ces opérations ;
- et savoir si un objet de ce type est modifiable ou non ; nous en reparlerons dans un chapitre ultérieur.

Remarque : De nos jours, la contrainte de la place mémoire est souvent moins importante que la contrainte de vitesse.

Les types principaux que nous manipulerons sont :

- Les entiers. Les calculs avec les nombres entiers sont exacts, et depuis Python 3 il n'y a pas de limite à la taille d'un entier.
- Les flottants (nombres à virgule) : là les calculs sont approchés, il peut donc y avoir des problèmes d'erreurs d'arrondi. En outre, il y a un maximum et un minimum pour les flottants manipulés. Ces limitations sont dues au fait qu'un flottant est toujours codé grâce à un nombre fixé de bits (en général 64 sur les ordinateurs actuels).
- Python propose aussi un type pour les nombres complexes (des flottants complexes), ce qui est rare dans les autres langages, et que nous utiliserons peu souvent.
- Les chaînes de caractères, pour enregistrer du texte.
- Les booléens : Vrai ou Faux (en Python `True` et `False`) qui interviennent notament dès qu'on effectue un test. Un booléen est un objet très simple et rapide à manipuler.
- Les tableaux : permettent d'enregistrer un nombre quelconque de données. Utilisé en permanence !

Remarque : Python est un langage assez souple : vous avez le droit d'effectuer des opérations mélangeant différent type, par exemple des entiers et des flottants (le résultat sera alors un flottant). De manière générale, Python essaie souvent de deviner ce que vous avez voulu dire quand vous tapez une instruction bizarre. Lorsqu'il devine mal, ceci crée des erreurs assez dures à détecter...

Exemple : `'a'*6`, `[1,2]*6`, `1+2.5`, `"abc"+"xyz"`...

Enfin, il existe des fonctions de conversion assez souples également. Par exemple les fonctions `int`, `float`, `str`, `list` transforment si possible l'objet passé en argument en entier, flottant, booléen ou tableau, respectivement.

La fiche de syntaxe Python distribuée en cours récapitule les principales fonctions à connaître sur ces différents types.

Nous étudierons plus en détails les entiers et flottants dans un chapitre ultérieur.

2 Booléens

Les booléens sont au nombre de deux : « vrai » et « faux ». En Python : `True` et `False`.

2.1 Tests

Les opérations telle que `==`, `<=` ou `<` renvoient des booléens. Et d'autre part lorsqu'on utilise un mot clé `if` ou `while`, Python recherche derrière un booléen.

Prenons l'exemple suivant :

```
1 def valeurAbsolue(x):
2     if x>0:
3         return x
4     else:
5         return -x
```

Suivons l'exécution de `valeurAbsolue(3)` :

- L'argument `x` prend donc la valeur 3. Ainsi la première ligne est évaluée en

  `if 3 > 0 :`

- Maintenant, Python exécute le programme chargé de comparer des nombres (c'est-à-dire la fonction noté `">"` ici). Celui-ci renvoie `True` car il est vrai que $3 > 0$. La première ligne est alors évaluée en :

  `if True :`

- Maintenant, Python s'occupe de lire le `if`. Voyant que ce `if` est suivi de `True`, il va exécuter le bloc en dessous, donc la ligne 2. Qui est évaluée en

  `return 3`

  il finit donc l'exécution du programme, en renvoyant la valeur 3.

2.2 Prédicats

Définition 2.1. *Un prédicat est une fonction dont l'ensemble d'arrivée est $\{vrai, faux\}$. Autrement dit, c'est une fonction qui renvoie un booléen.*

Par exemple, écrivons le prédicat `estPositif` qui prend en entrée un flottant x et qui renvoie le booléen qui indique si x est positif.

```
1 def estPositif(x):
2     """ Entrée : un entier ou un flottant x.
3         Sortie : le booléen x>0."""
4     if x>=0: return True
5     else: return False
```

Remarque :

- Lorsque les instructions à exécuter après un `if` ou un `else` sont très courtes, on n'est pas obligé d'aller à la ligne.
- On choisit souvent pour les prédicats des noms qui commencent par « `est` ».

Maintenant, nous pouvons réécrire la fonction valeur absolue à l'aide de ce prédicat :

```
1 def vAbs(x):
2     if estPositif(x):
3         return x
4     else:
5         return -x
```

On remarque que le code ainsi produit est particulièrement lisible. Et la lisibilité doit être votre préoccupation numéro 1.

Autre exemple : écrivons un prédicat `divise` qui prend deux entiers a et b en entrée et qui indique si $a|b$:

```
1 def divise(a,b):
2     if b%a==0:
3         return True
4     else:
5         return False
```

Maintenant, nous pouvons utiliser cette fonction pour écrire une autre fonction renvoyant « vrai » si un nombre est pair, et « faux » sinon :

```
def estPair(n):
    if divise(2,n):
        return True
    else:
        return False
```

Cependant, on peut faire beaucoup plus simple :

```
def estPair(n):
    return divise(2,n)
```

et à la réflexion pour la fonction divise :

```
def divise(a,b):
    return a%b == 0
```

ou pour la fonction `estPositif` :

```
def estPositif(x):
    return x>=0
```

2.3 Opérations sur les booléens

Il y a trois fonctions à connaître sur les booléens, qui sont « et », « ou », et « non ». En Python, `and`, `or`, et `not`.

- « non » est une fonction qui prend un booléen et en renvoie un autre, on peut l'écrire ainsi mathématiquement :

$$\begin{array}{rcl} \{vrai, faux\} & \to & \{vrai, faux\} \\ b & \mapsto & \left| \begin{array}{ll} vrai & \text{b} = faux \\ faux & \text{b} = vrai \end{array} \right. . \end{array}$$

- « et » et « ou » sont des opérations internes sur les booléens, qui prennent deux booléens et en renvoient un troisième (le terme précis est « loi de commutation interne, souvent abrégé en lci). Exactement comme les opérations + ou × sur les nombres. Vous connaissez déjà les tables d'addition et de multiplication, voici les tables de "et-ation" et de "ou-ation" [1]. Je note V et F au lieu de vrai et faux pour simplifier [2].

et	V	F
V	V	F
F	F	F

ou	V	F
V	V	V
F	V	F

Le « ou » est inclusif : si les deux côté d'un « ou » sont vrais, alors le résultat est vrai.

À titre d'exemple, suivons l'exécution du programme suivant :
cf exercice: 1
Voici les propriétés de ces trois opérations :

- Les lci « et » et « ou » sont associatives. En conséquence, il n'est pas nécessaire de mettre des parenthèses dans une expression utilisant uniquement une de ces deux opérations.
- Ces deux opérations sont commutatives à un « détail » près : lors de l'évaluation d'un « et » ou d'un « ou », Python va évaluer en premier le membre de gauche. Et si cette évaluation suffit à donner le résultat, alors il ne va pas du tout évaluer le terme de droite. On dit que l'évaluation des ces opérateurs est « paresseuse ». Le fait que le terme de droite ne soit pas évalué aura deux conséquences :
  - ◇ Gain de temps
  - ◇ Pas d'erreur au cas le membre de droite ne soit pas défini. Exemple : « `if x<>0 and 1/x < 2` fonctionne, alors que « `if 1/x <2 and x<>0` » déclenche une erreur lorsque `x` est nul.
- L'opération « ou » est distributive sur « et », mais « et » est également distributive sur « ou ».

1. Penser à demander à l'académie française de rajouter ces deux mots au dictionnaire du français.
2. En fait le symbole mathématique pour « vrai » est « ⊤ », et celui pour « faux » est « ⊥ ». Et tant qu'à indiquer les symboles mathématiques : « et » se note ∧, « ou » se note ∨, et « non » se note ¬.

- Vrai est élément neutre pour « et », et Faux est élément neutre pour « ou ».
- Enfin, les lois de De Morgan donnent le lien entre les deux lci « et » et « ou » et la fonction « non » :
  - $\diamond$ $\forall (a,b,c) \in \{V,F\}^3$, non ($a$ ou b) = (non a) et (non b) ;
  - $\diamond$ $\forall (a,b,c) \in \{V,F\}^3$, non ($a$ et b) = (non a) ou (non b).

3 Listes et tableaux

3.1 Introduction

Les listes et les tableaux sont utilisés lorsqu'on a de nombreuses données à sauvegarder.

Les listes (ou pile, la différence sera explicité dans un chapitre ultérieur) permettent de rajouter ou d'enlever des éléments, mais en contrepartie, l'accès à un élément au milieu de la liste est plus compliqué. Au contraire, les tableaux sont des structures qui doivent toujours contenir le même nombre de données ; en contrepartie l'accès à n'importe quel élément est presque immédiat (temps borné).

Plus précisément, voici les opérations possibles pour chacun de ces deux types :

Listes	Tableaux
• Créer une liste vide	• Créer un tableau d'une taille fixée
• Rajouter ou supprimer des éléments	• Lire et modifier un élément du tableau

Bien sûr, étant donné un tableau, si n est sa longueur, il est toujours possible de rajouter un élément : il suffit de créer un autre tableau d'une taille $n+1$, puis de recopier les n premières données du premier vers le second tableau. Mais cette opération prendra un certain temps... Alors que dans une liste, elle sera quasi instantanée.

Ainsi, le choix d'utiliser un tableau ou une liste permet d'optimiser la vitesse d'exécution selon le type d'opérations à faire. En gros :

- Si on sait le nombre de données à enregistrer : tableau
- Sinon : liste.

Cependant en Python, il n'y a pas vraiment de liste ni de tableau. À la place, il y a des « tableaux redimensionnables », appelés `list`. Il s'agit d'une structure de donnée qui combine les avantages des listes comme des tableaux : on peut facilement accéder aux éléments au milieu, et on peut aussi en rajouter à la fin ! Ainsi en programmation Python, on utilise le type `list` comme type à tout faire.

La contrepartie est que ces tableaux redimensionnables sont plus lents que de véritables tableaux ou de véritables listes.

Dans la suite, on se permettra éventuellement d'écrire « tableau », voire « liste » pour parler des tableaux redimensionnables de Python.

Enfin, signalons que la bibliothèque `numpy` propose de véritables tableaux. Ils sont appelés `array`. Il s'agit donc de structures de taille fixe, mais beaucoup plus rapide que les `list` du Python de base. Nous les utiliserons au second semestre pour du calcul numérique.

3.2 Principales commandes

Voici les principales commandes Python. On trie les commandes ci-dessous selon qu'elles se rapportent plutôt au point de vue liste ou tableau.

Attention : l'indexation des éléments d'un tableau commence à 0. Ainsi, si un tableau `T` est le longueur n, alors son premier élément est `T[0]`, et son dernier `T[n-1]`.

1. **Opérations de type liste :**
   - Créer une liste vide : `maListe=[]`.
   - Ajouter un élément x à la fin de la liste : `maListe.append(x)`.
   - Retirer le dernier élément de la liste : `maListe.pop()`.
     En outre, ceci renvoie l'élément enlevé : par exemple l'instruction `a=maListe.pop()` enlève le dernier élément de la liste et l'enregistre dans a.
   - Renvoyer la concaténation deux listes : `liste1 + liste2`.
2. **Opération de type tableau :**
   Attention : les éléments du tableau sont indicés à partir de 0. Ainsi, si le tableau `T` contient n éléments, le premier est `T[0]` et le dernier est `T[n-1]`. Notamment, `T[n]` renvoie une erreur.
   - longueur du tableau : `len(monTableau)`.

- accéder à l'élément d'indice i : `monTableau[i]`.
- modifier l'élément d'indice i : `monTableau[i]=nouvelle valeur`.

Remarque : Pour parcourir tous les éléments du tableau avec une boucle pour, on utilisera donc le plus souvent

```
for i in range(0, len(T)):
    # Faire quelque chose avec t[i]
```

On voit une certaine cohérence dans la syntaxe Python : pas de +1 ni de -1.

Exemple : Tirer au sort un élève, en supposant qu'on dispose d'un tableau contenant le nom de tous les élèves de la classe :

```
import numpy.random as rd
def eleveAuHasard(t):
    """ t est le tableau des élèves de la classe.
    Cette fonction en renvoie un hasard."""
    n=len(t)
    i=rd.randint(0,n)
    return t[i]
```

Et pour ceux qui aiment écrire le code le plus court possible :

```
def eleveAuHasard(t):
    """ t est le tableau des élèves de la classe.
    Cette fonction en renvoie un hasard."""
    return t[rd.randint(0, len(t))]
```

3.3 Exemples

3.3.1 Somme des éléments d'un tableau

Nous savons déjà calculer des sommes. La méthode s'adapte sans difficulté pour calculer la somme des éléments d'un tableau :

```
def somme(t):
    """ Renvoie la somme des éléments de t."""
    res = 0 # Variable qui va contenir la somme des éléments déjà lus
    n=len(t) #nb de cases dans t
    for i in range(0, n):
        res += t[i] # On ajoute l'élément i de t dans res
    return res
```

3.3.2 Recherche dans un tableau

Algorithme à connaître d'après le programme officiel.
Écrivons un prédicat qui indique si un tableau contient un certain élément.

C'est le moment d'indiquer un point de méthode. Lors de la conception d'un programme, il y a quelque soit ce programme, deux questions à se poser (au moins) :

- Comment gérer la répétition : boucle pour, boucle tant que, autre ?
- Quelles informations faut-il retenir, autrement dit, de quelles variables aurons-nous besoin ?

Répondons à ces deux questions dans la situation présente :

- On voit qu'il suffit de parcourir le tableau une fois, de gauche à droite, case après case. Par conséquent, le programme sera basé sur une simple boucle « `for i in range(0, len(t))` ».
- Nous allons utiliser une variable `trouvé` qui indiquera si nous avons trouvé l'élément cherché. Ainsi, cette variable contiendra un booléen. Initialement, elle vaudra `False`, et dès que (ou plutôt « si ») nous rencontrons l'élément cherché, elle passera à `True`.

```python
def appartient(x,t):
    """ Indique si x appartient à t."""
    trouvé=False
    for i in range(0, len(t)):
        if t[i]==x:
            trouvé=True
    return trouvé
```

Dans un second temps, on constate qu'on peut optimiser légèrement ce programme. En effet, au moment où nous trouvons l'élément cherché, nous pouvons nous arrêter : il est inutile de lire le reste du tableau.

Pour ce faire, la méthode académique[3] est d'utiliser une boucle « tant que ».

Concernant la condition de la boucle : nous voulons continuer à chercher tant que nous n'avons pas trouvé. Cependant, il faut également prendre garde à ne pas sortir du tableau !

```python
def appartient2(x,t):
    """ Indique si x appartient à t."""
    trouvé=False
    n=len(t)
    i=0
    while not trouvé and i<n:
        # Ici, trouvé est vrai ssi x est dans t[0:i]
        # Valeur maximale possible pour i à cet endroit : n-1. Donc t[i] existe.
        if t[i]==x:
            trouvé=True
        i+=1
        # Maintenant, trouvé est vrai ssi x est dans t[0:i]
    return trouvé
```

J'ai indiqué ce que contient la variable `trouvé`. On remarque que c'est la même formule logique en début et en fin de boucle, simplement i a augmenté de 1 entre temps.

3.3.3 Garder les éléments positifs

Voici un autre exemple de fonction élémentaire sur les tableau. Il va nous faire utiliser la fonction `append` qui permet de rajouter un élément. À propos : la syntaxe de `append` est particulière, puisqu'il faut taper *tableau*`.append(`*élément à rajouter*`)`. C'est une syntaxe propre à Python. Ce genre de fonction s'appelle une « méthode ».

Le but est d'écrire une fonction prenant en entrée un tableau et renvoyant un nouveau tableau contenant uniquement les éléments positifs du premier.

Il faudra créer un autre tableau, initialement vide, dans lequel on rajoutera au fur et à mesure les éléments positifs rencontrés.

```python
def elementsPositifs(t):
    """ Renvoie un tableau contenant les éléments positifs de t."""
    res=[]

    for i in range(0, len(t)):
        if t[i] >0:
            res.append(t[i])
    return res
```

3.3.4 Renverser un tableau

```python
def retourne(t):
    """ Renvoie les éléments de t mais dans l'ordre inverse."""

    res= []
    n=len(t)
    for i in range(0, n ):
```

3. Il est également possible d'utiliser un `return` pour arrêter le programme au milieu d'une boucle. Cet usage détourné du `return` n'est pas possible dans tous les langages de programmation, et engendre uns perte de lisibilité et un risque d'erreur, c'est pourquoi je ne la présente pas ici dans le cadre d'un cours pour débutant.

```
        res.append(t[n-1-i])
    return res
```

On peut proposer cette deuxième version qui consiste à parcourir le tableau en sens inverse. C'est l'occasion de présenter une variante du « `range` » : on peut ajouter un troisième argument, que l'on appelle le « pas », et qui indique de combien le compteur sera incrémenté à chaque étape. Par exemple dans un `for i in range(0,10,2)`, le compteur `i` prendre successivement les valeurs $0, 2, 4, 6, 8$. Pour faire une boucle à reculons, il suffit de mettre un pas de -1.

```
# Version 2
def retourne(t):
    """ Renvoie les éléments de t mais dans l'ordre inverse."""
    res= []
    n=len(t)
    for i in range( n-1, -1 , -1  ):
        res.append(t[i])
    return res
```

3.3.5 Tester si un tableau contient deux éléments identiques

C'est un exemple plus difficile : il nécessite deux boucles « pour ».

```
def contientUnDouble(t):
    """ Entrée : un tableau t
        Sortie : le booléen `` t contient deux éléments identiques'' """
    n=len(t)
    res= False # indique si on a trouvé un double

    for i in range(0, n):
        for j in range(i+1,n):
            if t[i] == t[j] :
                res=True

    return res

```

4 Chaînes de caractères

Les chaînes de caractères permettent d'enregistrer du texte. Leur importance vient du fait que le texte est la manière de communiquer avec un humain : d'une part, à la fin du calcul, le résultat devra être affiché, ce qui sous-entend de le mettre sous forme d'un texte d'une manière et d'une autre, et d'autre part lorsque l'humain voudra fournir des données à l'ordinateur, il lui fournira le plus souvent du texte, que ça soit directement en le tapant ou en créant un fichier de données.

Ainsi le texte sert-il pour l'interface entre l'homme et la machine. Mais il est également utile pour l'interface entre différents langages. Par exemple, si un programme en Python doit communiquer avec une base de données SQL (cf fin de l'année), il faudra à un moment donné écrire le texte d'une requête SQL dans le code Python. Au niveau du système d'exploitation également : pour passer des arguments depuis le système d'exploitation vers un programme, comme pour récupérer des données renvoyées par le programme, le plus simple est d'utiliser du texte.

Ceci s'explique car le type des chaînes de caractère est le seul qui soit (plus ou moins) universel pour tous les langages de programmation. Au contraire, les entiers, flottants, booléens, tableaux... sont encodé par chaque langage d'une manière qui lui est propre et donc ne peuvent être transférés de manière simple d'un langage à un autre.

Les chaînes de caractères sont donc un type plus important qu'on pourrait le croire au premier abord.

4.1 Commandes élémentaires

En Python, les opérations élémentaires ont la même syntaxe que pour les tableaux :

- créer une chaîne : `maChaine= "abc"` ou `maChaine='abc'`
- le mot vide : `""` ou `''`.
- longueur de la chaîne : `len(maChaine)`.
- accéder au caractère d'indice i : `maChaine[i]`

- concaténation de deux chaînes : `chaine1 + chaine2`.

Par contre, contrairement aux tableaux, il n'est pas possible de modifier une chaîne en place : une commande de type `maChaine[i] = 'x'` renvoie une erreur. On dit qu'une chaîne est "non mutable", ou "persistante".

Remarque : Un raccourci Python : si `t` est un tableau ou une chaîne, alors `t[a:b]` renvoie la liste ou la chaîne formée des éléments d'indice a jusqu'à $b-1$.

4.1.1 Exemples

4.1.2 Garder les voyelles

Écrivons une fonction prenant en entrée une chaîne de caractère et renvoyant la liste de ses voyelles.

C'est le même principe que l'exemple de 3.3.3.

Pour tester si une lettre `x` est une voyelle, on peut utiliser `appartient(x, ["a", "e", "i", "o", "u", "y"])`, où `appartient` est le programme que nous avons déjà écrit qui teste si un élément appartient à un tableau.

En réalité, ce programme est déjà présent dans Python : il suffit de taper `x in ["a", "e", "i", "o", "u", "y"]`.

```
def extraitVoyelles(m):
    """ Renvoie la liste des voyelles de m."""

    res=[] # Va contenir les voyelles de m qu'on a lues
    for i in range(0, len(m)):
        if appartient(m[i], tabVoyelles): #si m[i] est une voyelle
            res.append(m[i])
    return res
```

Remarque : En fait, comme la syntaxe des chaînes de caractères est la même que celle des tableaux, le programme `appartient` permet aussi de tester si une lettre appartient à une chaîne. Ainsi on peut remplacer « `appartient(x, ["a", "e", "i", "o", "u", "y"])` » par « `appartiente( x, "aeiouy")` ». Et en utilisant la fonction prédéfinie dans Python : « `x in "aeiouy"` ».

4.2 Recherche d'un mot dans un texte

Cet algorithme figure dans la liste des exemples classiques du programme officiel. A savoir refaire donc !

Soient t et m deux chaînes de caractères. Le but est de rechercher si m figure dans t. Penser par exemple que t est un grand texte, et m un simple mot. C'est l'algorithme qui est lancé dans la plupart des logiciels lorsqu'on tape ctrl+F (« F » pour « find »).

Pour simplifier la compréhension et l'écriture de l'algorithme, nous allons créer un algorithme intermédiaire `ChercheMotALaPlacei` qui étant donné un entier i regarde si le mot m est présent dans le texte t à la place i.

Nous prenons donc en entrée les chaînes de caractère t et m ainsi que l'entier i. Et nous renverrons un booléen. Nous allons parcourir toutes les lettres de m et vérifier à chaque fois si il y a la même lettre à la place correspondante dans t. Pour tout k entre 0 et longueur de m -1, il s'agit donc de voir si $m[k] = t[i+k]$.

Comme nous pouvons connaître à l'avance le nombre de lettres dans m et donc le nombre d'itération à faire, nous pouvons utiliser une boucle "pour". Cependant, il serait également pertinent d'utiliser une boucle "tant que", car on pourrait s'arrêter dès qu'une lettre de m diffère de la lettre correspondante dans t, ce qui fait gagner un peu de temps.

Nous allons utiliser une variable `résultat` de type booléen, qui devra donc à la fin de l'algorithme contenir Vrai si et seulement si m est dans t à la place i. Au départ, nous mettrons "Vrai" dans cette variable, et dès qu'une lettre diffère, nous passerons à "Faux".

Nous pouvons ajouter une petite vérification : pour pouvoir aller jusqu'au bout de m sans dépasser de t, il faut que longueur(m)+i <longueur(t)

Remarque : Il existe des algorithmes plus performant... En outre, dans la plupart des outils de recherche un peu évolué, il y a des options permettant de recherche un motif et non juste un mot. Par exemple, on peut vouloir rechercher un mot commençant par "inf" et finissant par "atique", ou un texte contenant un "@" et finissant par ".fr", ou une suite de 10 chiffres commençant par 06 ou 07, rechercher un nom de fichier contenant "thunderstruck" et finissant par ".mp3", etc...

Pour cela il existe des algorithmes pointus et efficaces, qui sont l'objet d'un des chapitres du programme de l'option informatique en MP.

ChercheMotALaPlacei :
Entrée : t et m chaînes de caractère, i entier
Sortie : un booléen, indiquant si m est présent dans t débutant à la place i.
Vérifier que longueur(m)+i <longueur(t)
résultat ← Vrai
pour *k de 0 à longueur de m-1* **:**
| **si** *m[k]≠t[k+i]* **:**
| | résultat ← Faux
| **fin**
fin
renvoyer résultat

ChercheMotALaPlacei : (version plus astucieuse)
Entrée : t et m chaînes de caractère, i entier
Sortie : un booléen, indiquant si m est présent dans t débutant à la place i.
Vérifier que longueur(m)+i <longueur(t)
résultat ← Vrai
pour *k de 0 à longueur de m-1* **:**
| résultat ← résultat et (m[k]=t[k+i])
fin
renvoyer résultat

ChercheMot :
Entrée : t et m chaînes de caractère
Sortie : un booléen, indiquant si m est présent dans t (à n'importe quelle place).
trouvé ← Faux
pour *i de 0 à longueur(t) - longueur(m) -1* **:**
| **si** *chercheMotALaPlacei(t, m, i)* **:**
| | trouvé ← Vrai
| **fin**
fin
renvoyer trouvé

ChercheMot : (version astucieuse)
Entrée : t et m chaînes de caractère
Sortie : un booléen, indiquant si m est présent dans t (à n'importe quelle place).
trouvé ← Faux
pour *i de 0 à longueur(t) - longueur(m) -1* **:**
| trouvé ← trouvé ou (chercheMotALaPlacei(t, m, i))
fin
renvoyer trouvé

4.3 Remarque : caractères spéciaux

Il existe certains caractères spéciaux, qui sont signalés par une barre contre-oblique (antislash en anglais). Voici les plus fréquents :

- Aller à la ligne : `\n` sous linux, `\r\n` sous windows, et `\r` sous mac.
- `\t` désigne la tabulation.
- Et pour afficher une contre-oblique ? C'est `\\`.

C'est dans les chemins d'accès aux fichiers sous windows que des problèmes se posent. En effet, windows utilise la contre-oblique pour séparer les dossiers dans une adresse de fichier. Par exemple `C:\nouveau dossier` posera problème car le `\n` sera interprété comme le caractère « nouvelle ligne ». Sous unix (mac ou linux) le caractère de séparation des dossiers est la barre oblique, il n'y a donc aucun problème.

Si vous vous demandez pourquoi Windows utilise deux caractères pour signaler la fin d'une ligne alors qu'un seul suffit : c'est un reste de l'époque des machines à écrire, ou à la fin d'une ligne il fallait :

- Ramener le chariot tout à droite (`\r` retour chariot) ;
- Faire monter la feuille d'un cran vers le haut (`\n` nouvelle ligne).

4.4 Encodage

La façon dont on enregistre un texte s'appelle l'encodage. A l'origine, chaque lettre était encodée sur 1 octet, donc 8 bits, ce qui fait 256 lettres possibles. Cet encodage s'appelle ASCII.

Mais les 256 lettres se sont vite avérées insuffisante lorsqu'on a voulu prendre en comptes les accents et de nombreux symboles spéciaux. Quasiment chaque pays a alors développé un (ou plusieurs) encodage(s) pour sa langue...

Vous avez sûrement déjà reçu un fichier texte qui à l'ouverture affiche des symboles ésotériques au lieu des lettres accentuées : c'est que la personne qui a créé le fichier n'utilise pas le même encodage que votre ordinateur.

Le dernier encodage en date est l'utf-8, il devrait être capable d'encoder tous les caractères de toutes les langues car il inclus la possibilité de rajouter des octets si les 8 initiaux ne suffisent pas. C'est l'encodage qu'on recommande à l'heure actuelle.

De nombreux éditeurs sont capables de détecter automatiquement l'encodage utilisé, cependant il est conseiller de l'indiquer explicitement dans vos fichiers. Dans Python ajouter la ligne suivante au début de vos fichiers :

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

5 Tableaux de tableaux (matrices)

Un tableau peut contenir n'importe quel objet, y compris d'autres tableaux.

Par exemple exécutons `A= [ [1,2,3], [5,2,10], [0,1,2]]`. Ainsi `A` est un tableau de tableaux. Que se passe-t-il lorsqu'on tape `A[1][2]` ?

- D'abord, `A[1]` récupère l'élément d'indice 1 de A. Il s'agit du tableau `[5,2,10]`.
- ensuite, `A[1][2]` récupère l'élément d'indice 2 de ce tableau. C'est donc 10.

On peut penser que `A` représente la matrice $\begin{pmatrix} 1 & 2 & 3 \\ 5 & 2 & 10 \\ 0 & 1 & 2 \end{pmatrix}$. Ainsi, `A[1][2]` donne l'élément de la ligne 1 et de la colonne 2 (toujours en comptant en partant de 0, ce qui ne sera pas le cas en math).

Exercice 1. * Tableau de notes

Supposons donné un tableau de tableaux `notes` contenant les notes des élèves d'une classe à différents devoirs. On note ne le nombre d'élèves, et nDS le nombre de devoirs. On suppose que pour tout $(i,j) \in [\![0, ne[\![\times [\![0, nDS[\![$, `notes[i][j]` donne la note de l'élève i pour le DS j. (Et on suppose que le professeur numérote ses élèves et ses DS à partir de 0)

1. Écrire une fonction pour calculer la moyenne d'un élève à l'année.
2. Écrire une fonction pour calculer la moyenne de la classe à un devoir donné.
3. Écrire une fonction pour calculer la moyenne de la classe à l'année.
4. Écrire une fonction pour calculer la note maximale sur tous les devoirs et tous les élèves.

Exercice 2. * fonctions élémentaires sur les matrices

Programmer les fonctions élémentaires suivantes, analogues à celles vues sur les tableaux simples :

1. Somme des éléments d'une matrice
2. Maximum des éléments d'une matrice

Deuxième partie

Exercices

TP du chapitre 2 : types de données

1 Booléens

Exercice 1. **! Années bissextiles

On rappelle qu'une année n est bissextile lorsque (n est multiple de 4) et (n n'est pas multiple de 100 ou n est multiple de 400).

commande utile : le reste de la division euclidienne de a par b s'obtient par `a % b`.

1. Programmer un prédicat qui indique si un entier n correspond à une année bissextile :
   (a) en utilisant les connecteurs logiques "et" et "ou" ;
   (b) avec trois tests imbriqués. Combien y-a-t-il de possibilités selon l'ordre des tests choisi ? En programmer deux, et dessiner l'arbre décisionnel correspondant.
2. En déduire une fonction `nbDeJours` qui prend en entrée un entier n et renvoie le nombre de jours de l'année n.
3. *Bonus :* Le calendrier Grégorien a été appliqué en France en 1582. Donc les années avant 1582 sont bissextiles dès qu'elles sont multiple de 4. Par ailleurs, 1582 elle même a perdu 10 jours (du 10 au 19 décembre inclus) pour compenser le retard accumulé par le calendrier julien. Corriger les question précédentes grâce à ces informations.

Exercice 2. * Théorème de Pythagore

1. Écrire un prédicat `estRectangleEnB` prenant en entrée les coordonnées de trois points A, B, C et indiquant si le triangle ABC est rectangle en B.
2. (**) Écrire un prédicat `estRectangle` prenant en entrée les coordonnées de trois points A, B, C et indiquant si le triangle ABC est rectangle.

Exercice 3. *! Exemples de prédicats : nombres premiers

1. (a) Écrire un prédicat `estPair` qui indique si un nombre entier est pair.
   (b) Écrire de même un prédicat pour indique si un nombre est multiple de 3. Utiliser les deux fonctions précédentes pour obtenir un prédicat qui indique si un nombre est multiple de 6.
2. (a) Écrire un prédicat `divise` prenant en entrée deux entiers a et b non nuls et indiquant si a divise b.
   (b) (**) Écrire un prédicat `estPremier` qui indique si un nombre entier est premier.

2 Tableaux

Exercice 4. *! Opérations élémentaires sur les tableaux

Ce genre de petites fonctions est utilisé en permanence en programmation.

1. Écrire une fonction qui renvoie la somme des éléments d'un tableau de nombres.
2. Écrire une fonction qui renvoie le maximum d'un tableau.
3. Modifier cette fonction pour qu'elle renvoie en outre l'indice pour lequel ce maximum est atteint ("l'argmax" du tableau).
4. Écrire un prédicat `estTrié` qui indique si les éléments d'un tableau sont dans l'ordre croissant.
5. Écrire une fonction `applique` qui prend en entrée un tableau `t` et une fonction f et qui renvoie le tableau obtenu en appliquant f à chaque élément de `t`.

   Par exemple si `carré` est la fonction carré, alors `applique( [1,2,3,4,5], carré)` renverra `[1,2,9,16,25]`

   Remarque : Ceci est déjà programmé sous Python. Pour appliquer la fonction `carré` à un tableau `t`, taper `list(map(( t, carré))`. Ou alors on peut utiliser `[carré(x) for x in t]`

Exercice 5. * Manipulation de tableaux plus difficile**

Chacun de ces programmes nécessitera deux boucles inconditionnelles.

1. Tester si un tableau contient deux éléments identiques.
2. Déterminer les deux éléments d'un tableau les plus proches l'un de l'autre.

Exercice 6. * « Pour tout » et « il existe »**

1. Écrire une fonction prenant en entrée un tableau et un prédicat et testant si tous les éléments du tableau vérifient le prédicat.

2. Même question avec "il existe" au lieu de "pour tout".
   Bonus : au lieu de programmer complètement cette fonction, on peut utiliser la fonction précédente.
3. Même question avec "il existe un unique".
4. Par exemple, en déduire une fonction qui teste si dans une liste d'entier, aucun n'est pair.
5. (***) Rédiger les trois fonctions précédentes de manière plus concise en n'utilisant aucun `if` mais uniquement les connecteurs logique `and` et `or`.

Exercice 7. ** ! Crible d'Ératosthène

Voici un algorithme très connu permettant étant donné un entier n de calculer tous les nombres premiers entre 2 et $n-1$.

Voici le principe : on commence par écrire tous les entiers jusqu'à $n-1$. Puis on barre les multiples de 2 sauf 2, puis les multiples de 3 sauf 3, etc. Au final, il ne reste que les nombres premiers.

1. Écrire une fonction `barre` prenant en entrée un tableau t d'entiers et un entier a et qui "barre" (en pratique qui mets 0) dans t toutes les cases d'indice multiple de a mais différent de a.
2. Écrire une fonction `initialisation` prenant en entrée un entier n et renvoyant un tableau contenant tous les entiers de 0 à $n-1$. Pour simplifier l'indiçage, faire en sorte que pour tout $i \in [\![0, n[\![$, `t[i]=i`.
3. Écrire une fonction `sansZero` prenant en entrée un tableau d'entier t et renvoyant un nouveau tableau contenant tous les nombres non nuls de t.
4. Écrire la fonction finale `Eratosthene` prenant en entrée un entier n et renvoyant la liste des nombres premiers entre 2 et $n-1$.

Exercice 8. * Diviseurs d'un entier**

Le but est d'écrire un programme calculant la décomposition d'un entier en facteurs premiers.

1. Écrire un prédicat `divise` prenant en entrée deux entiers a et b et renvoyant `true` si a divise b et `false` sinon.
   Commande python utile : le quotient de la division euclidienne s'obtient par `//`, et le reste par `%`.
2. Écrire un programme `diviseur` prenant en entrée un entier n et calculant le plus petit diviseur positif de n. Justifier mathématiquement que ce diviseur est premier.
3. Finalement, écrire un programme `decomposition` qui prend en entrée un entier n et qui renvoie sa décomposition en facteurs premiers. On renverra le résultat sous forme d'une liste.

Exercice 9. ** Rechercher le nombre manquant dans une liste (Centrale 2017)

Écrire une fonction qui prend en entrée un tableau t de longueur $n-1$ qui contient tous les entiers de $[\![0, n[\![$ sauf un, et qui renvoie cet entier manquant. On pourra rédiger au préalable une fonction `appartient` qui teste si un nombre est présent dans un tableau.

Combien de comparaisons sont effectuées dans le pire des cas ?

Exercice 10. * Plus grande plage de 0 consécutifs**

Écrire une fonction pour calculer le nombre maximal de zéros consécutifs dans un tableau.

3 Chaînes de caractère

Exercice 11. * Séparer les mots dans un texte**

Écrire une fonction qui découpe un texte en mots, c'est-à-dire qui découpe le texte selon ses espaces. Par exemple `decoupe("Arthur le glomorphe à rayure marche à vive allure")` renverra `["Arthur", "le", "glomorphe", "à", "rayures", "marche", "à", "vive", "allure"]`.

Exercice 12. * Ordre alphabétique**

1. Écrire une fonction prenant en entrée deux mots $m1$ et $m2$ et renvoyant `True` si $m1$ précède $m2$ dans l'ordre alphabétique [4], et `False` sinon.
2. Reprendre l'exercice du calcul du maximum (exercice 4) avec un tableau de chaînes de caractères.

Remarque : L'ordre alphabétique est déjà implémenté dans Python : c'est tout simplement l'opération `<` lorsqu'utilisée entre deux chaînes de caractères.

4. Le terme officiel est « ordre lexicographique »

4 Matrices

Exercice 13. *! Opérations élémentaires sur les matrices

1. Calculer la somme des coefficients d'une matrice
2. Calculer le plus grand des coefficients d'une matrice
3. Écrire une fonction prenant en entrée un entier n et renvoyant la matrice I_n, c'est-à-dire la matrice de format $n \times n$ contenant des 1 sur la diagonale et des 0 ailleurs.

Exercice 14. ** Calcul de moyenne

On suppose dans cet exercices que les notes des élèves de la classe aux différents devoirs du semestre sont enregistrés dans une matrice `notes`, de format $n \times p$ où n est le nombre d'élèves et p le nombre de devoirs. Ainsi, $\forall (i,j) \in [\![0,n[\![\times[\![0,p[\![$, `notes[i][j]` est la note de l'élève i au devoir j.

1. Écrire une fonction prenant en entrée une telle matrice `notes` et un entier $i \in [\![0,n[\![$ et renvoyant la moyenne de l'élève i.
2. Écrire une fonction prenant en entrée une telle matrice `notes` et un entier $j \in [\![0,p[\![$ et renvoyant la moyenne de la classe au devoir j.
3. Écrire une fonction prenant en entrée une telle matrice `notes` et renvoyant la moyenne de la classe sur tout le semestre.
4. Écrire une fonction pour calculer l'élève, puis le devoir ayant la meilleure moyenne.

Exercice 15. **! Manipulation de listes et chaînes de caractères : avec la liste des élèves

Le programme `extraire` fournit permet de récupérer un tableau contenant la liste des élèves. Chaque élément de ce tableau sera lui-même un tableau à deux éléments de la forme `["NOM", "prénom"]`.

1. Créer deux listes séparées, l'une contenant les prénoms, l'autre les noms.
2. Compter le nombre d'élèves dont le prénom commence par une voyelle.
3. Rajouter avant le nom le numéro de l'élève dans le tableau.
4. Repartir les élèves en trinômes. On essaiera de faire en sorte que le programme fonctionne quel que soit le nombre d'élèves dans la classe : il faudra éventuellement créer un binôme, voire un monôme.
5. Quelle expression permet de récupérer l'initiale du prénom du deuxième élève du quatrième trinôme ?

Quelques indications

3 Pour `estPremier` : utiliser une variable qui passera à `True` si on a rencontré un diviseur.

4 3) La seule petite difficulté technique est d'initialiser la variable qui contiendra le résultat. Moralement on voudrait l'initialiser à $-\infty$... Il est peut-être plus simple l'initialiser au premier élément du tableau.

6 Finalement, la structure est la même que pour le calcul d'une somme. Dans le fond, on remplace `+` par `et` pour un « pour tout » , et par `ou` pour un « il existe ».

Pour « $\exists !$ », une possibilité est d'utiliser deux variables booléennes : `existe` et `unique`.

7 Entre deux versions plus ou moins optimisées de `barre` et `Erathostène`, il peut y avoir de très grosses différences de performance !

10 Il va falloir deux variables : une pour retenir le nombre maximal de zéros consécutifs déjà rencontrés, et une autre pour le nombre actuel de zéros consécutifs rencontrés.

11 Une seul parcourt de la chaîne suffit. Par contre, il faut réfléchir un peu finement aux variables à utiliser. On propose :

- `res`, une liste de chaînes de caractères pour contenir tous les mots déjà rencontrés.
- `motEnCours`, une chaîne, qui contiendra les lettres déjà rencontrées du mot en cours.

Ainsi le principe est que lorsqu'on rencontre un espace, `motEnCours` contient un mot entier. On le met alors dans `res`, puis on le vide pour pouvoir commencer le mot suivant.

12 Analysez en détails ce que vous faites lorsque vous comparez deux mot dans votre tête... Essayez d'écrire une définition mathématique de l'ordre alphabétique.