INFORMATIQUE

PRÉPAS SCIENTIFIQUES

Chez le même éditeur

Python 3
2e édition
Bob Cordeau, Laurent Pointal
304 pages
Dunod, 2020

Informatique
Joëlle Delacroix *et al*.
480 pages
Dunod, 2017

Algorithmes
Thomas H. Cormen
240 pages
Dunod, 2013

Jean-Noël BEURY

j'intègre

INFORMATIQUE

PRÉPAS SCIENTIFIQUES

MÉTHODES ET EXERCICES

DUNOD

Conception et création de couverture : Hokus Pokus Créations

11 rue Paul Bert, 92240 Malakoff
www.dunod.com

ISBN 978-2-10-080798-7

Table des matières

Avant-propos

Présentation générale

Cet ouvrage de la série « Méthodes et exercices » traite de l'intégralité du programme d'informatique en classes préparatoires aux grandes écoles. Chacun des quinze chapitres est divisé en quatre parties.

Les méthodes à retenir

Chaque chapitre commence par des rappels de cours synthétiques, des méthodes de raisonnement avec des exemples de programmes.

Énoncés des exercices

Des énoncés d'exercices d'application du cours et de nombreux exercices d'écrits et d'oraux de concours sont proposés. Ils sont affectés d'un niveau de difficulté, de 1 à 4.

Du mal à démarrer ?

Des indications de méthode sont données pour le cas où cela vous serait nécessaire.

Corrigés des exercices

Les solutions détaillées sont entièrement rédigées. De nombreux commentaires sont ajoutés dans les programmes Python, en particulier les indices de début et de fin pour les boucles `for`.

« Plus en ligne »

Vous pouvez télécharger à partir de la page de présentation de l'ouvrage sur le site Dunod tous les programmes Python et les requêtes SQL des exercices. Des fichiers complémentaires sont également fournis afin de tester les programmes, par exemple des images pour l'exercice sur la stéganographie dans le traitement des images. Pour le dernier chapitre, des fichiers .txt sont fournis afin de créer facilement une base de données pour tester les requêtes SQL.

https://dunod.com/EAN/9782100807987

Conseils de travail

Les deux premiers chapitres permettent de vous approprier la syntaxe Python. Je vous conseille de les relire régulièrement pour bien comprendre le langage Python : typage des variables, indentation, boucle, test, fonction…

Avec une bonne maîtrise du langage Python, vous pouvez vous concentrer sur les consignes des problèmes de concours. Les énoncés demandent très souvent d'écrire des fonctions. Faites bien attention aux arguments d'entrée et arguments de sortie. Pensez à utiliser les fonctions définies dans les questions précédentes. L'optimisation des algorithmes est souvent demandée afin de réduire la complexité.

À l'écrit, vous ne pouvez pas tester vos programmes Python sur un ordinateur. Je vous conseille de prendre une feuille de brouillon et de vérifier les différentes étapes réalisées par les boucles `for`, en particulier la première et la dernière. Les indices des listes et des tableaux ne doivent pas dépasser une valeur limite.

Soyez bien vigilant à l'indentation. Et n'oubliez pas que le programme quitte immédiatement une fonction quand il rencontre l'instruction `return`.

Entraînez-vous régulièrement à écrire des programmes Python et des requêtes SQL !

J'espère que cet ouvrage vous aidera à réussir le mieux possible l'épreuve d'informatique des concours et je vous souhaite bon courage pour votre travail.

Types – Listes – Boucles – Fonctions

CHAPITRE 1

Plan

Thèmes abordés dans les exercices

- Types utilisés avec Python : `int`, `float`, `str`, `list`, `bool`, `tuple`
- Tests avec les opérateurs : `and`, `or`, `not`
- Bibliothèque `math`
- Boucles `for`, `while`
- Définition d'une fonction

Points essentiels du cours pour la résolution des exercices

- Ajouter, supprimer des éléments dans une liste
- Manipuler des listes de listes
- Bien identifier les indices de début et de fin dans les boucles
- Utiliser les fonctions définies précédemment dans des fonctions

Les méthodes à retenir

Installation de Python 3

Il existe de très nombreux programmes permettant d'installer Python 3 sur l'ordinateur. On peut utiliser par exemple la suite Anaconda. Le logiciel Spyder permet de créer des programmes avec l'extension `.py`. Il faut faire attention de ne pas installer Python 2. Les programmes écrits avec Python 2 ne sont pas compatibles avec Python 3.

Types utilisés pour les variables

Types `int` **(entier)**, `float` **(flottant)**, `str` **(chaîne de caractères)**

Pour affecter une valeur dans une variable, on utilise « = » :

```
a=3    # a prend la valeur 3
```

Cette instruction affecte 3 dans la variable `a`.

Le nom des variables ne doit pas commencer par un chiffre et ne doit pas comporter de tiret. Par contre, on peut utiliser « _ ». Exemple de variables :

```
v_exp=2    # v_exp est possible alors que v-exp ne peut pas représenter une
           # variable
b2=3       # la variable b2 prend la valeur 3. Par contre, la variable 2b
           # n'est pas autorisée
```

La fonction `print(a)` permet d'afficher la valeur de `a` :

```
print(a)
```

Cette instruction affiche la valeur de `a`.

Toutes les variables dans Python ont un type. La fonction `type(a)` retourne le type de la variable `a`.

```
print(type(a))  # affichage du type de la variable a
```

Le symbole `#` permet d'ajouter des commentaires. Les caractères après `#` font partie du commentaire dans le programme Python.

On obtient sur l'afficheur :

```
int
```

La variable `a` est de type `int`, c'est-à-dire entier.

```
b=5.2              # b est un nombre flottant dont la valeur est 5,2
print(type(b))     # affichage du type de b : float
```

Attention

On n'écrit pas `5,2` mais `5.2` avec Python.

On peut effectuer des calculs avec Python :

```
c=(a+b)/2     # calcul de la moyenne de a et b
d=3.1e-2      # 3,1e(-2)
e=float("infinity")     # flottant représentant l'infini
print(e)      # Python affiche inf
```

La variable `b` est de type `float`, c'est-à-dire un nombre à virgule flottante.

On peut définir le type `str` (chaîne de caractères) en mettant des apostrophes ou des guillemets autour du mot.

```
mot1="Bon"      # on aurait pu écrire 'Bon'
mot2='jour'     # on aurait pu écrire "jour"
mot3="C'est un mot"
# on ne peut pas mettre des apostrophes car mot3 contient une apostrophe
print(mot3)
```

On obtient sur l'afficheur :

```
"C'est un mot"
```

On peut afficher plusieurs éléments sur même ligne. Il suffit de les séparer par une virgule dans la fonction `print()`.

```
print("La moyenne de a et b est :",c)
```

On peut concaténer des chaînes de caractères :

```
mot4=mot1+mot2
print(mot4)
```

On ne peut pas additionner un entier et une chaine de caractères.

```
d=a+mot1
```

On obtient le message d'erreur :

```
TypeError: can only concatenate str (not "int") to str
```

On peut convertir une chaîne de caractères en réel ou en entier.

```
C1='123'     # C1 est chaîne de caractères de type str
a=int(C1)    # conversion de C1 en entier
C2='12.5'    # C2 est une chaîne de caractères
b=float(C2)  # conversion de C2 en float
c=a**3+b**2  # calcul de a*a*a + b*b
print(c)     # affichage de c
```

Type `list` (liste)

Une liste est une collection ordonnée d'éléments. On utilise les crochets pour définir des listes avec Python. On peut avoir des éléments de type différent dans une liste. Pour créer une liste contenant les éléments 3, 5 et 8, on utilise la syntaxe suivante :

```
L1=[3,5,8]
```

On peut créer une liste avec `range()` (voir paragraphe « Boucle `for` » pour la syntaxe de `range()`) :

```
L2=list(range(6))    # on obtient L2= [0, 1, 2, 3, 4, 5] liste de 6 éléments
```

On peut concaténer deux listes en utilisant « + » :

```
L1=L1+[6,'mot']    # concaténation de la liste L1 et de la liste [6,'mot']
print(L1)    # on obtient : [3, 5, 8, 6, 'mot']
```

On peut utiliser également la fonction `append()`.

```
L1.append(9) # on ajoute l'élément 9 à la liste L1
print(L1)    # on obtient : [3, 5, 8, 6, 'mot', 9]
```

Dans les programmes Python, on privilégiera l'utilisation de `append()`.

On peut supprimer le dernier élément ajouté dans la liste et récupérer sa valeur :

```
x=L1.pop()   # x prend la valeur 9 de type int
print(L1)    # on obtient : [3, 5, 8, 6, 'mot']
```

On peut supprimer un élément quelconque d'une liste :

```
del L1[2]    # on supprime l'élément d'indice 2
print(L1)    # on obtient [3, 5, 6, 'mot']
```

Pour créer une liste vide :

```
L3=[]        # création d'une liste vide
L3.append([3, 2])    # la liste L3 vaut : [3, 2]
```

La fonction `len()` permet d'obtenir le nombre d'éléments dans une liste :

```
n=len(L1)    # la variable n vaut 4
```

On peut concaténer une liste *n* fois avec elle-même :

```
L4=[3,2,1]*4 # retourne [2, 1, 2, 1, 2, 1, 2, 1]
             # la liste [3,2,1] est concaténée 4 fois avec elle-même
```

Attention

« * » ne fait pas la multiplication par 4 de chaque élément de cette liste.

`L.sort()` trie la liste `L` en place, dans l'ordre croissant de ses éléments :

```
print(L4)    # [3, 2, 1, 3, 2, 1, 3, 2, 1, 3, 2, 1]
L4.sort()    # cette fonction ne retourne rien. Elle trie L4 en place
print(L4)    # [1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3]
```

On verra dans le chapitre 12 différents algorithmes pour trier les listes.

Indices des listes avec Python

On définit une liste `L5` :

```
L5=[-3,8.2,5,19]
n=len(L5)    # la longueur de la liste vaut n = 5
```

Remarque

Les indices des listes contenant *n* éléments sont numérotés de 0 à `n`–1 dans Python, contrairement à Matlab et Scilab, où les listes sont numérotées de 1 à `n`.

Pour obtenir le premier élément de la liste :

```
a=L5[0]     # la variable a prend la valeur -3
```

Pour obtenir le troisième élément de la liste `L5` :

```
b=L5[2]     # b vaut 5
```

Attention, le troisième élément de la liste `L5` a pour indice 2.

On a deux possibilités pour obtenir le dernier élément de la liste `L5` :

```
n=len(L5)
c=L5[n-1]   # valeur de l'élément d'indice n-1, c'est-à-dire le dernier
            # élément de L5
d=L5[-1]    # valeur du dernier élément de L5
```

On peut modifier un élément d'une liste :

```
L5[1]=25    # on modifie la valeur de l'élément d'indice 1
print('Liste L5 :',L5)    # sur l'afficheur, on a :
                          # "Liste L5 : [-3, 25, 5, 19]"
```

Listes de listes

```
L=[]      # création d'une liste vide
L.append([3,2,1])    # ajout du premier élément de la liste
L.append([8,6,4])    # L vaut [[3, 2, 1], [8, 6, 4]]
```

Chaque élément de la liste est une liste.

Pour extraire le premier élément de la liste :

```
M=L[0]    # M vaut [3,2,1]
```

Pour récupérer le deuxième élément de `M` :

```
a=M[1]    # a vaut 2
```

On peut également écrire :

```
b=L[0][1] # b vaut 2
```

Type `bool` (booléen)

On peut définir une variable qui vaut vrai ou faux. Avec Python, on utilise la syntaxe `True` ou `False`.

```
Rep1=True   # Rep1 vaut True. Le type est bool (booléen)
Rep2=False  # Rep2 vaut False. Le type est bool (booléen)
```

On verra au chapitre 2 le type `array`, permettant de définir des tableaux.

Type `tuple` (tuple)

Un tuple ressemble à une liste mais les éléments ne peuvent pas être modifiés une fois qu'ils sont créés. On utilise « `()` » pour définir un tuple alors qu'on utilise « `[]` » pour définir une liste.

```
L=(2, 5, 2)  # création d'un tuple contenant trois éléments
a=L[0]       # a vaut la valeur du premier élément du tuple
```

On peut récupérer les valeurs des éléments du tuple. On peut modifier la valeur d'un élément d'une liste mais on ne peut pas le faire pour un tuple.

```
L[1]=8    # message d'erreur :
          # "TypeError: 'tuple' object does not support item assignment"
```

Python renvoie un message d'erreur indiquant que l'on ne peut pas modifier la valeur de `L[1]`.

Sauf indication contraire, on n'utilisera pas les tuples dans les programmes. Par contre, on verra que les fonctions renvoient des tuples et on cherchera à récupérer les différents éléments du tuple (voir exercice 1.2 « Recherche du minimum dans une liste non triée »).

On peut définir un tuple en omettant les parenthèses :

```
L2=3, 6, 1         # on définit un tuple
print(type(L2))    # le type est un tuple
```

Permutation de deux variables

On a très souvent besoin dans les programmes de permuter deux éléments.

On peut écrire

```
x=3
y=5
```

```
c=x
x=y    # x vaut 5
y=c    # y vaut 3
```

On peut utiliser la syntaxe `u,v=v,u` qui permute `u` et `v` :

```
x=3
y=5
x,y=y,x    # on obtient x=5 et y=3
```

On peut réduire le programme à deux lignes :

```
x,y=3,5    # x=3 et y=5
x,y=y,x    # on obtient x=5 et y=3
```

La première ligne définit un tuple qui vaut (3,5).

La deuxième ligne définit un tuple en fonction d'un autre tuple, d'où la permutation des deux éléments.

Opérateurs arithmétiques

`+`	addition
`-`	soustraction
`*`	multiplication
`/`	division
`//`	division entière
`%`	modulo ou reste
`**`	exponentiation ou puissance
`abs()`	valeur absolue
`round(x)`	renvoie la valeur de l'entier le plus proche de *x*. Si deux entiers sont équidistants, l'arrondi se fait vers la valeur paire
`round(x,2)`	arrondit *x* avec une précision de 2 chiffres après la virgule

```
x=2**4             # x vaut 16=2*2*2*2
q=9//2             # le quotient vaut 4
r=9%2              # le reste vaut 1
a=abs(-9.2)        # a vaut 9.2
b=round(8.5163,2) # arrondi avec 2 chiffres après la virgule : 8.52
c=(a+b)/x          # calcul avec des parenthèses
```

Tests

On utilise la syntaxe **if** (« si », en français) pour faire un test. Les opérateurs de comparaison sont :

`==`	égal à
`!=`	différent de
`<`	strictement inférieur
`<=`	inférieur ou égal
`>`	strictement supérieur
`>=`	supérieur ou égal

Il faut ajouter « : » à la fin de la ligne.

Les lignes devant être exécutées si la condition est vérifiée doivent toutes être indentées.

```
L2=[18.2, 9.5, 15]
note=L2[1]
if note==10:              # ne pas oublier ":" à la fin de la ligne
    print("Note = 10")    # indentation pour exécuter cette ligne si note=10
```

On crée une liste `L2` contenant 4 éléments. On récupère la deuxième valeur de la liste `L2` dans la variable note. On teste si cette note vaut 10, alors on obtient sur l'afficheur :

```
Note = 10
```

On peut utiliser la syntaxe `if…else`.

```
if note==10:              # ne pas oublier ":" à la fin de la ligne
    print("Note = 10")    # indentation pour exécuter cette ligne si note=10
else:    # ne pas oublier ":" à la fin de la ligne
    print('Note différente de 10')
# indentation pour exécuter cette ligne si note est différent de 10
```

On peut chercher à effectuer plusieurs tests si le premier n'est pas vérifié. Dans ce cas, on utilise la syntaxe `elif` qui signifie « sinon si ». Si le test précédent n'est pas vérifié, il effectue un nouveau test et ainsi de suite.

```
if note==10:              # ne pas oublier ":" à la fin de la ligne
    print("Note = 10")    # indentation pour exécuter cette ligne si note=10
elif note <=5:
    print('Note inférieure à 5')
    print(note)           # affichage sur une autre ligne
elif note<=15:
    print('Note inférieure de 15 :',note)    # affichage sur la même ligne
else:
    print('Note > 15')
```

Les tests sont effectués dans l'ordre du programme. Si une condition est vérifiée, les tests ultérieurs avec `elif` ne sont pas effectués.

On peut avoir des opérateurs logiques dans les tests.

```
a==b         # renvoie un booléen True si a=b et False sinon
Rep==True    # renvoie un booléen True si Rep=True et False sinon
```

Si on veut réunir des expressions booléennes, on peut utiliser les opérateurs : `and` (intersection), `or` (union), `not` (négation).

```
a==a         # renvoie un booléen True
not(a==a)    # renvoie un booléen False
```

Remarque

On peut utiliser `&` à la place de `and`.

```
a, b=3, 2    # affectation sur une seule lignes des variables a et b
Rep=False    # la variable Rep est un booléen
if (a==b) and (Rep==True):    # teste si a=b et si Rep=True
    print(a)
else:
    print(a+b)
```

Bibliothèque `math`

La bibliothèque `math` permet d'ajouter de nouvelles fonctions mathématiques : `cos()`, `sin()`, `tan()`, `exp()`, `sqrt()` (racine carrée), `log()` (logarithme népérien), `log10()` (logarithme décimal), `pi` (nombre Pi)…

Il faut ajouter le nom de la bibliothèque pour utiliser les nouvelles fonctions :

```
import math  # bibliothèque math
a=math.pi/4
b=math.cos(math.pi/4)
c=math.sin(math.pi)
print(b)     # affiche 0.7071067811865476
print(c)     # affiche 1.2246467991473532e-16
```

Dans certains problèmes de concours, on utilise la syntaxe suivante :

```
from math import *    # bibliothèque math
a=pi/4
b=cos(pi/4)
```

Lorsqu'on utilise la syntaxe `from math import *`, il n'est plus nécessaire d'ajouter le nom de la bibliothèque pour utiliser les fonctions de cette bibliothèque.

Or on utilise très souvent d'autres bibliothèques (`numpy`, `random`…) et certaines fonctions portent le même nom dans des bibliothèques différentes.

Il est donc préférable de ne pas utiliser `from math import *` mais plutôt `import math`. On peut renommer la bibliothèque `math` en `m` par exemple :

```
import math as m    # bibliothèque math renommée en m
a=m.pi/4
```

On peut être surpris que Python n'affiche pas 0 lors du calcul de $\sin(\pi)$. Le type `float` ne permet pas le calcul exact mais une valeur approchée avec une précision d'environ 10^{-16}.

Pour effectuer le test « $\sin(x)=0$ », on n'utilisera pas :

```
sin(x)==0    # retourne False si x = pi !
```

mais le programme suivant :

```
eps=1**-8          # on choisit une valeur pour eps
abs(sin(x))<eps    # retourne True si x = pi
```

Boucle `for`

Les boucles permettent de répéter des opérations un certain nombre de fois. On utilise la syntaxe `for i in range()`.

```
for i in range(3): # i varie de 0 inclus à 3 exclu avec un pas égal à 1
    b=i*2+3        # indentation pour exécuter cette ligne à chaque étape
    print(b)       # indentation pour exécuter cette ligne à chaque étape
```

`range(3)` signifie que la boucle `for` va être exécutée 3 fois.

Il faut bien indenter les lignes devant être exécutées à chaque étape de la boucle `for`.

Ne pas oublier d'écrire « `:` » à la fin de la ligne `for`.

i prend successivement les valeurs 0, 1 et 2 :

- Première étape de la boucle `for` : `i` = 0. On obtient « 3 » sur l'afficheur.
- Deuxième étape de la boucle `for` : `i` = 1. On obtient « 5 » sur l'afficheur.
- Troisième étape de la boucle `for` : `i` = 2. On obtient « 7 » sur l'afficheur.

On ne retrouve pas la syntaxe « `end` » comme dans d'autres langages. La fin de l'indentation représente la fin de la boucle `for`.

Syntaxe de la fonction `range()`

```
for i in range(start, stop, step):
```

`i` est un entier qui varie de `start` inclus à `stop` exclu avec un pas égal à `step` :

- `start` : indice de départ inclus ;
- `stop` : indice final exclu ;
- `step` : cette variable désigne le pas.

Remarque

Ne pas confondre avec la fonction `linspace(mini, maxi, nbr_pts)`, où les éléments sont compris entre `mini` inclus et `maxi` inclus (voir chapitre 2). `nbr_pts` désigne le nombre de points.

```
for i in range(n):    # Python prend par défaut : start = 0 et step = 1
```

`i` varie entre 0 inclus et `n` exclu avec un pas égal à 1.

La boucle `for` sera donc exécutée `n` fois.

Parcours des éléments d'une liste ou d'une chaîne de caractères

```
L=[3, 5, 8]      # liste L contenant 3 éléments
for elt in L:    # elt prend successivement la valeur d'un élément de L
    print(elt)   # affichage d'un élément de L à chaque étape
```

`elt` prend à chaque étape la valeur d'un élément de `L`.

La boucle `for` sera exécutée 3 fois.

Boucle `while`

Lorsqu'on ne connaît pas à l'avance le nombre d'étapes dans une boucle, on utilise des boucles `while` (tant que).

Il faut bien indenter les lignes devant être exécutées à chaque étape de la boucle `while`.

```
i=0
while i!=3:    # ne pas oublier ":" à la fin de la ligne while
    i=i+1      # on incrémente i de 1 à chaque étape
    b=i*2+3    # b est calculé à chaque étape
    print(b)   # b est affiché à chaque étape
```

Remarque

On peut utiliser la syntaxe suivante :

```
i+=1    # la variable i est incrémentée de 1
```

- La variable `i` est initialisée à 0.
- Première étape de la boucle `while` (`i` est différent de 3) : `i` est incrémenté de 1 et vaut 1. Python affiche « 5 ».
- Deuxième étape de la boucle `while` (`i` est différent de 3) : `i` est incrémenté de 1 et vaut 2. Python affiche « 7 ».
- Troisième étape de la boucle `while` (`i` est différent de 3) : `i` est incrémenté de 1 et vaut 3. Python affiche « 9 ».

À la fin de la troisième étape, `i` vaut 3. La condition « *i* différent de 3 » n'est plus vérifiée. Le programme quitte la boucle `while`.

Définition d'une fonction

On utilise l'instruction `def` pour définir une fonction dans Python. Dans la parenthèse, on indique les arguments d'entrée.

Le corps de la fonction est indenté d'un niveau. Les objets définis dans le corps de la fonction sont locaux à la fonction (voir paragraphe « Passage par valeur, passage par référence, variable globale, variable locale dans les fonctions » dans le chapitre 2).

Une fonction peut renvoyer des objets avec l'instruction `return`. Le programme quitte immédiatement la fonction quand il rencontre l'instruction `return`.

Exemple de fonction

On souhaite définir la fonction $f(t) = 3\cos(5t)$.

```
from math import *  # bibliothèque math
def f(t):           # argument d'entrée : t
    y=3*cos(5*t)    # calcul de y. Ne pas oublier le symbole *
    return y        # argument de sortie : y
```

On peut utiliser cette fonction dans le programme principal. L'utilisateur tape au clavier la valeur de `t`. On obtient une chaîne de caractères que l'on convertit en `float`. Il reste à appeler la fonction pour obtenir le résultat dans la variable `res`.

```
rep=input("Taper la valeur de t :")     # rep est une chaîne de caractères
T=float(rep)  # conversion de rep en float
res=f(T)      # le résultat de la fonction est affecté dans la variable res
print("f("+rep+")=", res)
```

Remarque

On pourrait écrire également :

```
t=float(input("Taper la valeur de t :"))    # rep est une chaîne de caractères
print("f("+rep+")=",f(t))
```

Si on écrit `return val1, val2, val3`, on récupère un tuple de 3 éléments après l'appel de cette fonction. Voir exercice 1.2 « Recherche du minimum dans une liste non triée » pour l'utilisation de ces données.

Énoncés des exercices

1.1

■□□□

Moyenne et écart-type de notes

a) Proposer une fonction `moyenne` qui admet comme argument d'entrée une liste non vide de notes. Cette fonction retourne la moyenne des notes.

b) Proposer une fonction `ecart_type` qui admet comme argument d'entrée une liste non vide de notes. Cette fonction retourne l'écart-type des notes.

1.2

■□□□

Recherche du minimum dans une liste non triée

On considère `L` une liste non vide et non triée contenant des éléments de type `float`.

a) Proposer une fonction `rec_min` qui admet comme argument une liste `L`. Cette fonction retourne le minimum des éléments.

b) Proposer une fonction `rec_min_ind` qui admet comme argument une liste `L`. Cette fonction retourne l'indice et la valeur du minimum de `L`.

c) Écrire le programme principal permettant d'afficher l'indice et la valeur du minimum pour la liste L2 = [3, 8.2, 5, 19, –2].

1.3

■■□□

Fonction factorielle

Proposer une fonction itérative `fact` qui admet comme argument un entier naturel n et retourne la factorielle de n.

1.4

■■■□

Calcul des termes d'une suite

On considère $(u_n)_{n\in\mathbb{N}}$ la suite des nombres définie par :

$u_0 = 1$, $\forall n \in \mathbb{N}$, $u_{n+1} = 3u_n + 5$.

a) Proposer une fonction `F1` qui admet comme argument un entier naturel n et retourne u_n. On n'utilisera pas de liste dans la fonction.

b) Proposer une fonction `F2` qui admet comme argument un entier naturel n et retourne la liste $L = [u_0, u_1, u_2, \ldots, u_n]$.

1.5

■■■■

File de voitures et codage binaire (Mines Ponts 2017)

On considère un ensemble de voitures sur une file à sens unique. Une file `L` de longueur n est représentée par n cases. Une case peut contenir au plus une voiture. Lorsqu'une case d'indice `i` de la file est occupée par une voiture, `L[i]` vaut `True`, sinon `L[i]` vaut `False`. Afin de comparer plus efficacement les files

représentées par des listes de booléens, on remarque que ces listes représentent un codage binaire où `True` correspond à 1 et `False` à 0.

a) Soit `L` une liste représentant une file de longueur n et i un entier tel que $0 \leq i < n$. Définir en Python la fonction `occupe(L, i)` qui renvoie `True` lorsque la case d'indice i de la file est occupée par une voiture et `False` sinon.

b) Combien existe-t-il de files différentes de longueur n ? Justifier votre réponse.

c) Écrire la fonction `versEntier(L)` prenant une liste de booléens en paramètre et renvoyant l'entier correspondant. Par exemple, l'appel `versEntier([True, False, False])` renverra 4.

d) On veut écrire la fonction inverse de `versEntier`, transformant un entier en une liste de booléens. Que doit être au minimum la valeur de `taille` pour que le codage obtenu soit satisfaisant ? On suppose que la valeur de `taille` est suffisante. Quelle condition booléenne faut-il écrire en quatrième ligne du code ci-dessous ?

```
def versFile(n, taille):
    res=taille*[False]
    i=taille-1
    while ...:
        if (n%2) !=0:    # % est le reste de la division entière
            res[i]=True
        n = n//2         # // est la division entière
        i = i - 1
    return res
```

1.6

Recherche d'un indice dans une liste (Mines Ponts 2018)

On s'intéresse à des mesures de niveau de la surface libre de la mer. On appelle *horodate* un ensemble (fini) des mesures réalisées sur une période de 20 minutes à une fréquence d'échantillonnage de 2 Hz. Les informations de niveau de surface libre (déplacement vertical en m) sont stockées dans une liste de flottants `liste_niveaux`. On suppose qu'aucun des éléments de cette liste n'est égal à la moyenne.

a) Proposer une fonction `moyenne` prenant en argument une liste non vide `liste_niveaux` et retournant sa valeur moyenne.

b) Proposer une fonction `ind_premier_pzd(liste_niveaux)` retournant, s'il existe, l'indice du premier élément de la liste tel que cet élément soit supérieur à la moyenne et l'élément suivant soit inférieur à la moyenne. Cette fonction devra retourner « −1 » si aucun élément vérifiant cette condition n'existe.

c) Proposer une fonction retournant l'indice i du dernier élément de la liste tel que cet élément soit supérieur à la moyenne et l'élément suivant soit inférieur à la moyenne. Cette fonction devra retourner « −2 » si aucun élément vérifiant cette condition n'existe. On cherchera à proposer une fonction de complexité en $O(1)$ dans le meilleur des cas.

1.7

Analyse de vagues (Mines Ponts 2018)

On s'intéresse à des mesures de niveau de la surface libre de la mer. On appelle *horodate* un ensemble (fini) des mesures réalisées sur une période de 20 minutes à une fréquence d'échantillonnage de 2 Hz. Les informations de niveau de surface libre (déplacement vertical en m) sont stockées dans une liste de flottants

`liste_niveaux`. On suppose qu'aucun des éléments de cette liste n'est égal à la moyenne. On considère que la mesure de houle est représentée par un signal $\eta(t) \in \mathbb{R}, t \in [0, T]$. On appelle niveau moyen m la moyenne de $\eta(t)$ sur [0, T]. On définit Z_1, Z_2, …, Z_n l'ensemble (supposé fini) des Passages par le Niveau moyen en Descente (PND) (voir figure 1). À chaque PND, le signal traverse la valeur m en descente. On suppose $\eta(0) > m$ et $\dfrac{\mathrm{d}\eta}{\mathrm{d}t}(0) > 0$. On en déduit que $\eta(t) - m \geq 0$ sur $[0, Z_1]$. Les hauteurs de vague H_i sont définies par les différences :

$$\begin{cases} H_1 = \max\limits_{t \in [0, Z_1]} \eta(t) - \min\limits_{t \in [Z_1, Z_2]} \eta(t) \\ H_i = \max\limits_{t \in [Z_{i-1}, Z_i]} \eta(t) - \min\limits_{t \in [Z_i, Z_{i+1}]} \eta(t) \text{ pour } 2 \leq i < n \end{cases}$$

On définit les *périodes de vagues* par $T_i = Z_{i+1} - Z_i$. La fréquence d'échantillonnage est égale à 2 Hz.

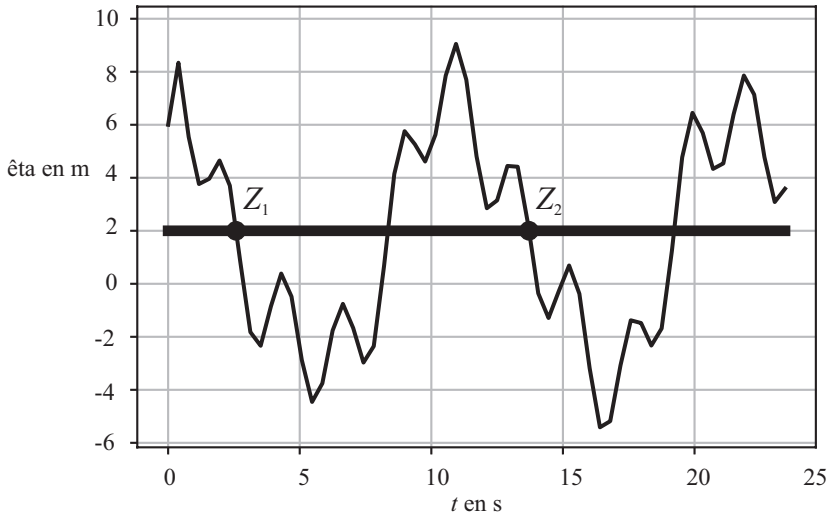

Figure 1 Passages par le Niveau moyen en Descente (PND). Ici, la moyenne m vaut 2.

On souhaite stocker, dans une liste `successeurs`, les indices des points succédant (strictement) aux PND (voir figure 2).

a) On propose la fonction `construction_successeurs`, qui retourne la liste `successeurs`. Compléter les lignes 6 et 7.

```
1  def construction_successeurs(liste_niveaux):
2      n=len(liste_niveaux)
3      successeurs=[ ]
4      m=moyenne(liste_niveaux)
5      for i in range(n-1):
6          if              # À compléter
7                          # À compléter
8      return successeurs
```

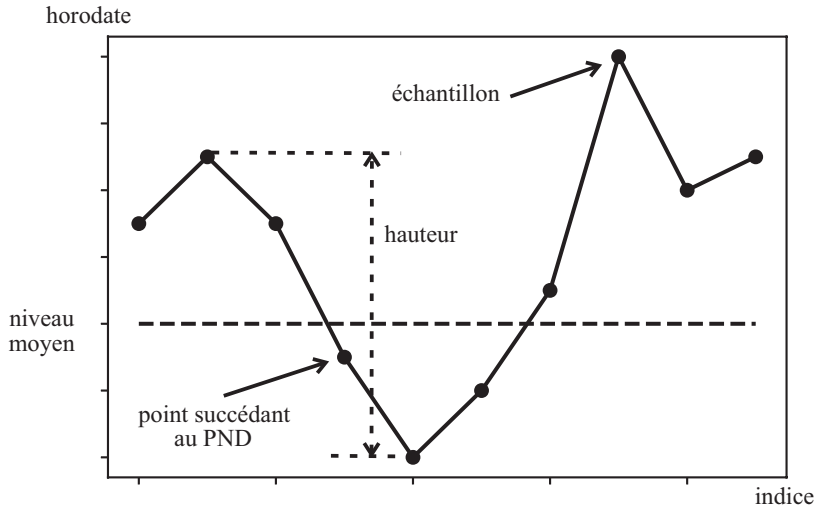

Figure 2 Propriétés d'une vague

b) Proposer une fonction `decompose_vagues(liste_niveaux)` qui permet de décomposer une liste de niveaux en liste de vagues. On omettra les données précédant le premier PND et celles succédant au dernier PND. Ainsi `decompose_vagues([1, -1, -2, 2, -2, -1, 6, 4, -2, -5])` (noter que cette liste est de moyenne nulle) retournera « [–1, –2, 2],[–2, –1, 6, 4] ».

c) On désire maintenant caractériser les vagues. Ainsi, on cherche à concevoir une fonction `proprietes(liste_niveaux)` retournant une liste de listes à deux éléments `[Hi, Ti]` permettant de caractériser *chacune des vagues i* par ses attributs :

- `Hi`, sa hauteur en mètres (m) (voir figure 2) ;
- `Ti`, sa période en secondes (s).

Proposer une fonction `proprietes (liste_niveaux)` réalisant cet objectif. On pourra utiliser les fonctions de Python `max(L)` et `min(L)` qui retournent le maximum et le minimum d'une liste `L`, respectivement.

d) Plusieurs indicateurs sont couramment considérés pour définir l'état de la mer. Parmi eux, on note :

- $H_{\max}$: la hauteur de la plus grande vague observée sur l'intervalle d'enregistrement [0, T] ;
- $H_{1/3}$: la valeur moyenne des hauteurs du tiers supérieur des plus grandes vagues observées sur [0, T] ;
- $T_{H\,1/3}$: la valeur moyenne des périodes du tiers supérieur des plus grandes vagues observées [0, T].

Proposer une fonction prenant en argument la liste `liste_niveaux` et retournant $H_{\max}$.

1.8

Loi de Poisson et loi binomiale

Soient n un entier naturel strictement positif et p un réel compris entre 0 et 1. On considère X et Y deux variables aléatoires à valeurs dans $\mathbb{N}$ sur un espace probabilisé donné. X suit une loi de Poisson de paramètre $\lambda = np$ et Y suit une loi binomiale de paramètres (n, p).

a) Proposer une fonction `Px`, d'arguments k, n et p, renvoyant la valeur de :

$$P(X = k) = \frac{\lambda^k}{k!}\exp(-\lambda)$$

$k!$ (factorielle de k) s'obtient par `math.factorial(k)` de la bibliothèque `math`.

Écrire le programme principal permettant d'afficher la liste des valeurs de $P(X = k)$ pour $k \in \mathbb{N}, 0 \leq k \leq 30$, avec $n = 30$ et $p = 0{,}1$.

b) Proposer une fonction `Py`, d'arguments k, n et p, renvoyant la valeur de :

$$P(Y = k) = \binom{n}{k} p^k (1-p)^{n-k} = \frac{n!}{k!(n-k)!} p^k (1-p)^{n-k}$$

Écrire le programme principal permettant d'afficher la liste des valeurs de $P(Y = k)$ pour $k \in \mathbb{N}, 0 \leq k \leq 30$, avec $n = 30$ et $p = 0{,}1$.

c) Soit $k \in \mathbb{N}$. On rappelle que, sous certaines conditions sur n et p, la probabilité $P(Y = k)$ peut être approchée par $P(X = k)$. Proposer une fonction `Ecart` d'arguments n et p, renvoyant le plus grand des nombres $\left|P(Y = k) - P(X = k)\right|$, pour $0 \leq k \leq n$.

d) Soit e un réel strictement positif. Proposer une fonction `N`, d'arguments e et p, renvoyant le plus petit $0 \leq k \leq n$ entier n tel que `Ecart(n, p)` soit inférieur ou égal à e.

e) Écrire le programme principal permettant d'afficher le résultat de la fonction `N` dans les deux cas suivants :

- $e = 0{,}008$; $p = 0{,}075$;
- $e = 0{,}005$; $p = 0{,}075$.

1.9

Nombres premiers (Mines Ponts 2019)

Le crible d'Ératosthène est un algorithme qui permet de déterminer la liste des nombres premiers appartenant à l'intervalle $[\![1, n]\!]$. Son pseudo-code s'écrit comme suit (algorithme 1 : crible d'Ératosthène) :

Données : N, entier supérieur ou égal à 1

Résultat : *liste_bool*, liste de booléens

début

liste_bool ← liste de N booléens initialisés à Vrai ;

Marquer comme Faux le premier élément de *liste_bool* ;

pour *entier* $i \leftarrow 2$ à $\left\lfloor \sqrt{N} \right\rfloor$ **faire**

si i n'est pas marqué comme Faux dans liste_bool **alors**

Marquer comme Faux tous les multiples de i différents de i dans *liste_bool* ;

fin

fin

retourner *liste_bool*

fin

À la fin de l'exécution, si un élément de *liste_bool* vaut Vrai alors le nombre codé par l'indice considéré est premier. Par exemple pour N = 4, une implémentation Python du crible renvoie [False True True False].

a) Sachant que le langage Python traite les listes de booléens comme une liste d'éléments de 32 bits, quelle est (approximativement) la valeur maximale de N pour laquelle *liste_bool* est stockable dans une mémoire vive de 4 Go ?