INFORMATIQUE AVEC PYTHON

MPSI·PCSI·PTSI
MP·PC·PSI·PT·TSI·TPC

Jean-Noël Beury

INFORMATIQUE AVEC PYTHON

MPSI-PCSI-PTSI
MP-PC-PSI-PT-TSI-TPC

j'intègre

MÉTHODES & EXERCICES

2e édition

DUNOD

Couverture : création Hokus Pokus, adaptation Studio Dunod

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

ISBN 978-2-10-084124-0

Table des matières

Avant-propos

Présentation générale

Cet ouvrage de la série « Méthodes & Exercices » traite de l'intégralité du programme d'informatique commune aux classes préparatoires aux grandes écoles. Chacun des seize 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 chapitre 15, des fichiers .txt sont fournis afin de créer facilement une base de données pour tester les requêtes SQL.

https://dunod.com/EAN/9782100841240

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, représentation graphique…

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 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`, `deque`, `dict`
- Tests avec les opérateurs : `and`, `or`, `not`
- Bibliothèques et modules
- Boucles `for`, `while`
- Définition d'une fonction
- Variable locale, variable globale

Points essentiels du cours pour la résolution des exercices

- Compteur, accumulateur
- Ajouter, supprimer des éléments dans une liste
- Création d'une liste par compréhension
- Slicing ou extraction de tranche
- Manipuler des listes de listes
- Bien identifier les indices de début et de fin dans les boucles
- Utiliser les fonctions définies précédemment dans d'autres 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 le logiciel Pyzo, qui permet de créer des programmes avec l'extension `.py`. Les programmes écrits avec Python 2 ne sont pas compatibles avec Python 3.

Script Python

Un script Python est formé d'une suite d'instructions. Une instruction simple est contenue dans une seule ligne. Si une instruction est trop longue pour tenir sur une ligne ou si on souhaite améliorer la lisibilité du code, le symbole \ en fin de ligne permet de poursuivre l'écriture de l'instruction sur la ligne suivante.

Types de base utilisés dans Python

Types `int` (entier), `float` (flottant)

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 typage des variables est dynamique : l'interpréteur détermine le type à la volée lors de l'exécution du code. Dans l'exemple précédent, le type de `a` est `int`.

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)
```

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

Opérations de base sur les entiers (`int`) : `+, -, *, //, **, %`

```
n=28
n//10    # 2=quotient de la division euclidienne de n par 10
n%10     # 8=reste de la division euclidienne de n par 10
n**3     # n puissance 3
```

Opérations de base sur les flottants (`float`) : `+`, `-`, `*`, `/`, `**`

```
a=1/3     # a vaut 0.3333333333333333
2.6**(a)  # 2,6 puissance a
```

Type `str` (chaîne de caractères)

Les chaînes de caractères (type `str`) sont des structures indicées immuables. On ne peut pas modifier les éléments d'une chaîne de caractères. On ajoute des apostrophes ou des guillemets autour d'un mot.

```
mot1= "C'est "   # mot1 est une chaîne de caractères de type str
n=len(mot1)      # n=6 (nombre de caractères)
mot2= 'un mot'   # on aurait pu écrire "un mot"
mot3=mot1+mot2   # concaténation de chaînes. On obtient "C'est un mot"
print(mot3[0])   # on obtient la première lettre de mot3
```

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 chaîne 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
```

On peut concaténer une chaîne de caractères *n* fois avec elle-même :

```
C1='ab'
C2=C1*3     # retourne 'abcabcabc'. Répétition de 'ab'
            # C1 est concaténée 3 fois avec elle-même
```

Type `bool` (booléen)

On peut définir une variable qui vaut `True` (vrai) ou `False` (faux).

```
rep1=True   # rep1 vaut True. Le type est bool (booléen)
rep2=False  # rep2 vaut False. Le type est bool (booléen)
a=12
b=10
```

```
rep3=(a==12)and(b==20)  # False pour le test : a=12 et b=20
rep4=not (a==13)        # rep4=True, on pourrait écrire : rep4 = a!=13
rep5=(a==12)or(b==20)   # True pour le test : a=12 ou b=20
```

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 l'instruction suivante :

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

Création d'une liste par compréhension :

```
L2=[i**2 for i in range(6)]   # on obtient L2= [0, 1, 4, 9, 16, 25]
```

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]
```

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']
```

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]*3                    # retourne [3, 2, 3, 2, 3, 2]. Répétition de [3, 2]
                              # la liste [3, 2] est concaténée 3 fois avec elle-même
```

On peut concaténer une liste avec une autre liste :

```
L5=L4+[1, 9]                  # on obtient : [3, 2, 3, 2, 3, 2, 1, 9]
```

Attention

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

La fonction `sort()` serait rappelée dans un problème de concours.

```
L5.sort()      # trie la liste L5 en place, dans l'ordre croissant de ses éléments
               # cette fonction ne retourne rien
print(L5)      # [1, 2, 2, 2, 3, 3, 3, 9]
```

On verra dans le chapitre 10 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 = 4
```

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
```

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

On peut représenter une matrice 2×3 par une liste `L` contenant 2 listes de longueur 3. Chacune de ces listes de longueur 3 représente une ligne de la matrice $\begin{pmatrix} 3 & 2 & 1 \\ 8 & 6 & 4 \end{pmatrix}$.

```
L=[[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 `tuple` (**tuple**)

Les tuples sont des structures indicées immuables. 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
n=len(L)     # n vaut le nombre d'éléments du tuple L
```

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 4.1 « Recherche du minimum dans une liste non triée » dans le chapitre 4 « Algorithmes, recherche dans une liste, recherche par dichotomie »).

On peut définir un tuple en omettant les parenthèses :

```
L2=3, 6, 1         # on définit un tuple
print(type(L2))    # le type est tuple
L3=L+L2            # concaténation de deux tuples : L3=(2, 5, 2, 3, 6, 1)
L4=(2, 1)*3        # création avec répétition : L4=(2, 1, 2, 1, 2, 1)
```

Type `deque` (deque)

Une deque (se prononce « dèque ») permet d'ajouter et de supprimer très rapidement des éléments aux extrémités droite et gauche (voir exercice 6.4 « Utilisation des deques » dans le chapitre 6 « Pile, file, deque, dictionnaire »). Pas d'extraction de tranche (ou slicing) avec les deques.

```
from collections import deque  : module permettant d'utiliser les deques
D=deque()             # création d'une deque vide D
D.append(8)           # ajoute 8 à l'extrémité droite de D
D.appendleft(5)       # ajoute 5 à l'extrémité gauche de D
x=D.pop()             # supprime l'élément à l'extrémité droite de D
x=D.popleft()         # supprime l'élément à l'extrémité gauche de D
D1=deque([3, 8, 5])
for elt in D1:        # affichage de tous les éléments de la deque D1
    print(elt)
```

Type `dict` (dictionnaire)

Voir exercice 6.5 « Comptage des éléments d'une liste à l'aide d'un dictionnaire » dans le chapitre 6 pour l'utilisation des dictionnaires.

Extraction de tranche ou slicing pour les listes, tuples et chaînes de caractères

Lorsqu'on veut extraire des éléments d'une liste, d'un tuple ou d'une chaîne de caractères, on utilise la technique du slicing ou extraction de tranche. Il suffit de mettre entre crochets les indices correspondant au début et à la fin de la « tranche » et de les séparer par « : ». On utilise la syntaxe :

```
A[start:stop]
```

- `start` : indice de depart, inclus.
- `stop` : indice final, exclu.
- `stop – start` : nombre d'éléments de la liste `A` que l'on souhaite extraire (avec un pas de 1 par défaut).

Liste

```
A=[3, 5, 1, 8, 10]     # liste de 5 éléments
```

Pour extraire les éléments [5, 1, 8] :

```
A[1:4]     # renvoie [5, 1, 8]
```

L'indice de départ vaut 1 avec `A[1]` = 5.

L'indice final vaut 4 – 1 = 3 avec `A[3]` = 8.

On récupère bien 3 éléments : [5, 1, 8].

Pour avoir un slicing avec un pas différent de 1, on utilise la syntaxe suivante :

```
A[start:stop:step]
```

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

On peut omettre `start`, `stop` ou `step`.

```
L=[2, 5, -3, 9, -4, 3]   # définition d'une liste
L1=L[:3]   # L1 contient les éléments d'indice i<3 : [2, 5, -3]
L2=L[3:]   # L2 contient les éléments d'indice i>=3 : [9, -4, 3]
L3=L[:]    # L3 contient tous les éléments de L
```

Tuple, chaîne de caractères

On utilise la même technique pour les tuples et les chaînes de caractères.

```
L=(3, 5, 8, -2)   # tuple
L2=L[0:3]         # L2=(3, 5, 8)
c="C'est un mot"  # chaîne de caractères
c2=c[0:3]         # c2="C'e"
```

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 l'instruction `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
`//`	quotient de la division euclidienne
`%`	reste de la division euclidienne
`**`	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            # quotient de la division euclidienne de 9 par 2
r=9%2             # reste de la division euclidienne de 9 par 2
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 <10:
    print('Note strictement inférieure à 10')
    print(note)           # affichage sur une autre ligne
elif note<=15:
    print('Note inférieure à 15 :',note)
else:
    print('Note strictement supérieure à 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)
```

Importation de modules

Des fonctions traitant d'un même domaine sont regroupées dans des modules (par exemple les fonctions mathématiques `cos`, `sin`, `tan`… sont regroupées dans le module `math`). Différents modules peuvent être regroupés dans une bibliothèque. On utilise l'instruction `import module` pour importer un module.

Module math

```
import math  # importation du module math
dir(math)    # liste des fonctions du module math
```

Le module `math` contient des fonctions et des variables : `cos()`, `sin()`, `tan()`, `exp()`, `sqrt()` (racine carrée), `log()` (logarithme népérien), `log10()` (logarithme décimal), `pi` (nombre π)…

Pour avoir plus d'informations sur une fonction ou une variable :

```
?math.pi     # Type : float. String form : 3.141592653589793
?math.cos    # Return the cosine of x (measured in radians)
```

Pour utiliser les fonctions et les variables du module `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
```

On peut être surpris que Python n'affiche pas 0 lors du calcul de sin(π). Le type `float` ne permet pas le calcul exact mais une valeur approchée avec une précision d'environ 10^{-16} (représentation des flottants sur des mots de taille fixe).

Pour effectuer le test `sin(x)==0`, on n'utilisera pas l'instruction :

```
m.sin(x)==0          # retourne False alors que x=pi
```

mais les instructions suivantes :

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

Lorsqu'on utilise l'instruction `from math import *`, il n'est plus nécessaire d'ajouter le nom du module pour utiliser ses fonctions :

```
from math import *   # module math
a=pi/4
```

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 le module `math` en `m` par exemple :

```
import math as m     # module math renommé m
a=m.pi/4
```

On peut importer des fonctions et des variables d'un module :

```
from math import cos, sin, tan, pi
a=cos(pi/4)
```

Module random

Les fonctions du module `random` seraient rappelées dans un problème de concours.

```
import random as rd  # module random renommé rd
n=20
i=rd.randint(0, n)   # indice aléatoire i compris entre 0 inclus et n-1 inclus
M=rd.random()        # nombre flottant aléatoire M tel que 0<=M<1
```

Importation de bibliothèques

La bibliothèque `matplotlib` regroupe plusieurs modules concernant les graphiques. L'instruction suivante permet d'importer le module `pyplot` de la bibliothèque `matplotlib` que l'on renomme `plt` :

```
import matplotlib.pyplot as plt  # module matplotlib.pyplot renommé plt
```

On peut avoir la liste des fonctions du module `pyplot` :

```
dir(plt)
```

L'instruction suivante permet de représenter `y` en fonction de `x` en utilisant la fonction `plot()` du module `pyplot` (voir chapitre 2 « Représentation graphique ») :

```
plt.plot(x, y)
```

On utilisera la bibliothèque `PIL` dans le chapitre 9 « Matrices de pixels et images » :

```
from PIL import Image            # module Image de la bibliothèque PIL
```

En physique, on utilise la bibliothèque `numpy`.

Boucle `for`

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

```
for i in range(3): # i varie entre 0 inclus et 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.

```
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` (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
```

La variable `i` est initialisée à 0 et incrémentée de 1 à chaque étape de la boucle `while`. On l'appelle un compteur.

Si la variable `i` est incrémentée d'une valeur différente de 1 ou décrémentée, on l'appellera un accumulateur.

Remarque

On peut utiliser l'instruction 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`.

L'instruction `break` fait sortir d'une boucle `while` ou `for` et passe à l'instruction suivante (voir exercice 10.8 « Tri à bulles » dans le chapitre 10 « Algorithmes de tri ». Lorsqu'il y a plusieurs boucles imbriquées, l'instruction `break` ne fait sortir que de la boucle la plus interne.

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 (touche TAB ou 4 espaces). Les variables définies dans le corps de la fonction sont des variables locales.

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 *  # module 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 suivant. 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 4.1 « Recherche du minimum dans une liste non triée » pour l'utilisation de ces données.

Objets muables, objets immuables

Il existe deux types d'objets dans Python :

- Les objets dont la valeur peut changer sont dits muables (ou *mutable* en anglais) : listes, dictionnaires, deques (voir chapitre 6 « Piles, files, deques, dictionnaire »)…
- Les objets dont la valeur ne peut pas changer sont dits immuables (ou *immutable* en anglais) : entiers, flottants, booléens, chaînes de caractères, tuples…

```
L1=[1, 2, 3, 4]
```

Cette affectation (ou assignation) est une instruction qui réalise les opérations suivantes :

- Création d'un objet muable (appelé `obj1`) de type `list` à une adresse mémoire. Cet objet possède un identifiant (adresse mémoire), un type et une valeur. La valeur de `obj1` vaut : `[1, 2, 3, 4]`.
- Création de la variable `L1`.
- Association de la variable `L1` avec l'objet `obj1` contenant la valeur `[1, 2, 3, 4]`.

Contrairement à d'autres langages de programmation (C ou Java), une affectation dans Python est une association d'une variable avec un objet contenant la valeur. C'est le choix des concepteurs du langage Python.

```
L2=L1
```

L'instruction `L2=L1` n'affecte pas `[1, 2, 3, 4]` à `L2` mais réalise les opérations suivantes :

- Création du nom de variable `L2`.
- Affectation à la variable `L2` de la référence (ou adresse mémoire) où est stocké `[1, 2, 3, 4]`.

`L1` et `L2` font donc référence au même objet `[1, 2, 3, 4]`.

La copie est très rapide puisqu'on n'occupe pas deux fois plus de place mémoire.

Si on modifie `[1, 2, 3, 4]` via `L1`, alors cette modification sera également visible par `L2`.

```
L1[0]=10
```

On constate que `L2[0]` vaut 10 également. C'est tout à fait normal car `L1` et `L2` font référence à la même adresse mémoire de `[10, 2, 3, 4]`.

On ajoute un élément dans `L1` avec la fonction `append` :

```
L1.append(12)
```

L'élément 12 est ajouté dans `L1` et `L2` puisque `L1` et `L2` font référence à la même liste modifiable (ou muable) : `[10, 2, 3, 4, 12]`.

Copie superficielle, copie profonde

```
import copy        # module copy
L1=[1, 2, [3, 4], 5]
L2=L1
L3=copy.copy(L1)
L4=copy.deepcopy(L1)
L1[0]=12
L1[2][0]=30
print('L1 =', L1)  # L1=[12, 2, [30, 4], 5]
print('L2 =', L2)  # L2=[12, 2, [30, 4], 5]
print('L3 =', L3)  # L3=[1, 2, [30, 4], 5]
print('L4 =', L4)  # L4=[1, 2, [3, 4], 5]
```

- La fonction `copy()` du module `copy` réalise une copie superficielle. Les éléments sont copiés s'il n'y pas de structure imbriquée. Si les éléments sont des listes par exemple, alors l'adresse mémoire des listes est copiée.
- La fonction `deepcopy()` du module `copy` réalise une copie profonde pour les structures imbriquées. Si les éléments sont des listes, alors la copie profonde copie bien les listes imbriquées.

Passage par valeur, passage par référence, variable globale, variable locale dans les fonctions

Une variable locale existe uniquement dans la table des variables d'une fonction. Elle est créée à l'appel de la fonction et détruite à la fin de la fonction.

Une variable globale est définie en dehors d'une fonction et peut être utilisée et modifiée dans la fonction. Il faut utiliser `global` pour déclarer une variable globale dans une fonction.

Lorsqu'une expression fait référence à une variable à l'intérieur d'une fonction (variable `d` par exemple), Python cherche la valeur définie à l'intérieur de la fonction et à défaut la valeur dans l'espace global du programme.

```
a=3
b=3
d=5
x=10
L=[3, 5, 8]   # définition de la liste L
print('Liste L',L)
def calcul(x, L):
    global a  # a est une variable globale
    x=2*x     # la valeur de x est multipliée par 2
    a=4       # la variable globale a est modifiée
              # en quittant la fonction calcul, on retrouve a = 4
    b=4       # cette variable est locale
    # en quittant la fonction, on ne retrouve pas cette valeur
    # car b=4 est défini localement dans la fonction calcul
    c=3       # cette variable est locale
    # cette variable existe uniquement dans la fonction calcul
    print(d)  # Python affiche 5
    L[0]=2
    # en quittant la fonction, les variables locales b et c sont détruites
    return a+b+c+x
print('Résultat de la fonction :',calcul(x, L))  # affiche 31
print(L) # la liste a été modifiée car L est passé par référence : [2, 5, 8]
print(a) # la variable globale a été modifiée : affiche 4
print(b) # b n'a pas été modifiée par la fonction calcul : affiche 3
print(x) # affiche 10
print(c) # erreur car c n'existe que dans la fonction calcul
```

Dans Python, les listes, les dictionnaires, les deques sont passés par référence et non par valeur.

L'argument d'entrée `L` de la fonction `calcul` fait référence à la liste `[3, 5, 8]` qui est un objet muable. La liste `L` est passée par référence. Lorsqu'on modifie `L` dans la fonction `f`, on modifie la liste `[3, 5, 8]`. La variable `L` dans la fonction `f` fait référence à la même adresse mémoire que la variable `L` dans le programme principal.

Il est donc inutile d'écrire `return(L)` à la fin de la fonction `calcul`. La fonction modifie la liste `L` et on retrouve la modification après avoir quitté la fonction.

On distingue deux cas importants pour les arguments d'entrée :

- Objet muable (liste, dictionnaire, deque…) : toute modification de cet objet dans la fonction est visible en dehors de la fonction. C'est « l'effet de bord » puisque la fonction modifie des données définies hors de sa portée locale.
- Objet immuable (entier, nombre flottant, booléen, chaîne de caractères, tuple…) : toute modification de cet objet dans la fonction n'est pas visible en dehors de la fonction. Pour simplifier, tout se passe comme si ces objets étaient passés par valeur.

Types des arguments dans les fonctions

Certains sujets de concours (Centrale-Supélec en particulier) utilisent des annotations en précisant les types des arguments et du résultat des fonctions.

On considère une fonction `F1` qui admet pour arguments d'entrée un entier `n`, un nombre flottant `x`, une chaîne de caractères `c1` et une liste `L`. Elle retourne un entier, un réel et une liste.