Fondamentaux

1 Programmer : concepts fondamentaux

1.1 Algorithmes et programmes

1.1.1 Rappels

Algorithme

Algorithme 1 : Test de parité

Données : $a \in \mathbb{N}^+$
Résultat : afficher si a est pair ou impair

```
1 a←valeur saisie par l'utilisateur
2 si reste(a/2)=0 alors
3 |  afficher a est pair
4 sinon
5 |  afficher a est impair
```

python

```
Test_parite.py
"""
Le programme prendra une valeur saisie
par l'utilisateur
et affichera s'il est pair ou impair.
"""
a=int(input("Saisir un entier positif"))
if a%2==0 :
    print("a est pair")
else :
    print("a est impair")
```

1.1.2 Fonctions

```python
def test(x):
    y=2*x
    return y

a=3
print(test(a+2))
```

1.1.3 Portée des variables

1.1.4 Paradigmes de programmation

1.2 Poser le problème et rédiger sa description

1.2.1 Les questions à se poser

- Définir et paramétrer les données du problème
- Décrire le résultat

1.2.2 La signature

La signature d'un algorithme, d'un programme ou d'une fonction est l'association de :

- Nom
- Arguments et leur type
- Type de la (des) valeur(s) du (des) résultat(s)

1.2.3 La spécification

La spécification d'un algorithme, d'un programme ou d'une fonction est l'expression du problème que peut être résolu. Elle qui contient :

- La signature
- Pré-condition sur les données (arguments, etc.) – leur type, les intervalles, les propriétés à respecter.
- Post-condition sur les résultats

1.2.4 Mise en œuvre en python

Spécification La spécification se rédige au début du programme ou de la fonction, entre triple guillements, en langage naturel le plus explicitement possible. Ces informations forment le `DocString`.

```python
def randint(self, a, b):
    """Return random integer in range [a, b], including both end points."""


def compresser (nomFichierEntree, nomFichierSortie):
    """Préconditions : nomFichierEntree est le nom d'un fichier contenant une image.
    Postconditions : un nouveau fichier nommé comme spécifié dans nomFichierSortie est
    créé, son contenu correspond àl'image initiale compressée"""
```

Signature Dans le cadre de cours, nous retiendrons la notation suivante, qui permet de préciser en commentaire le type des arguments et le type du résultat.

```python
def test(x:int)->int:
    return 2*x

#Version CCS
def maFonction(n:int, X:[float], c:str, u) -> (int, np.ndarray):
    ...
```

1.2.5 Vérifier les conditions

Pré-condition : le mot clé `assert` permet de vérifier une condition nécessaire au fonctionnement du programme. Si la condition n'est pas vérifiée, l'exécution s'interrompt. Par exemple :

- Vérification du signe d'un nombre : `assert a>=0`
- Vérification du type d'une variable `a` : `assert type(a)==int`
- Ou aussi : `assert isinstance(a,int)`

Post-condition :

1.2.6 Application

```python
"""
Programme exemple : finalité purement pédagogique.
"""

from math import sqrt

help(sqrt)
print('vérification : ')
print(sqrt(4))
print(sqrt(-4)) #????

```

```python
def racines_trinome_dans_R(a:float,b:float,c:float)->(list):
    """   Parameters
    a : float
    b : float
    c : float
    Returns
    -------
    (list) la liste des racines reelles du polynome : ax2+bx+c """
    assert a!=0, 'a doit être non nul' # sinon l'algorithme est inadapté au problème
    D=b**2-4*a*c
    if D>0:
        return [(-b-sqrt(D))/2/a,(-b+sqrt(D))/2/a]
    elif D<0:
        return []
    else:
        return [-b/2/a]

print(racines_trinome_dans_R(0,1,1)) #????
print(racines_trinome_dans_R(1,0,-1))
print(racines_trinome_dans_R(1,0,1))
print(racines_trinome_dans_R(1,2,1))
print(racines_trinome_dans_R(1,2*2**.5,2))
print(racines_trinome_dans_R(1,-2*3**.5,3))
```

1. Entourer le ou les import(s) de fonctions ;
2. Entourer le ou les docstring ;
3. Entourer la ou les signatures ;
4. Justifier chacun des affichages générés par ce programme.

2 Instructions d'un programme

2.1 Instructions

Instruction

Variable

Typage dynamique

Affectation

2.2 Les types de base et leur codage en machine

L'information de base utilisé par un système numérique est le bit, qui ne peut prendre que deux valeurs : 0 ou 1. Cette information élémentaire constitue le type `bool`.

2.2.1 Codage des entiers : type `int`

0 et 1 sont les deux chiffres d'un système binaire. Un entier naturel n s'écrira alors avec $p+1$ chiffres binaires.

$$\forall n \in \mathbb{N}^*, n = (a_p \dots a_0)_2 = \sum_{i=0}^{p} a_i 2^i.$$

$(a_p \dots a_0)_2$ est l'écriture binaire du nombre n. En informatique, on utilise aussi la base héxadécimale.

$$\forall n \in \mathbb{N}^*, n = (b_q \dots b_0)_{16} = \sum_{i=0}^{q} b_i 16^i.$$

En python, les entiers sont codés sur au moins 64 bits.

2.2.2 Codage des nombres à virgules : type `float`

On code un nombre à virgule flottante x de manière générale sous la forme (Norme IEEE 754) :

$$x = (-1)^s(1+m)2^{e-d}$$

Dans cette écriture, s est appelé le signe, m la mantisse et e l'exposant.

- $s \in \{0,1\}$ est toujours codé sur un bit, qui est le bit de plus fort poids ;
- $e - d \in \mathbb{Z}$ est codé sur les bits suivants, comme entier naturel « décalé » d'un certain nombre, variable suivant le type de codage et sur un nombre de bits également variable ;
- $m \in [0,1[$ est codé sur les bits de plus faible poids.

s	e_{10}	$\cdots$	e_0	m_{-1}	$\cdots$	m_{-52}

En python, les flottants sont codés sur 64 bits. Il faut retenir :

- le plus grand float : 2.10^{308} ;
- le plus petit positif non nul : $\approx 2.10^{-308}$;
- la précision relative : $2^{-55} \approx 10^{-16}$

2.2.3 Codage d'un caractère

Pour coder un caractère latin, nous retiendrons la table d'équivalence entre un entier et un caractère. Cette table s'appelle ASCII.

0	<NUL>	32	<SPC>	64	@	96	`	128	Ä	160	†	192	¿	224	‡
1	<SOH>	33	!	65	A	97	a	129	Å	161	°	193	¡	225	·
2	<STX>	34	"	66	B	98	b	130	Ç	162	¢	194	¬	226	‚
3	<ETX>	35	#	67	C	99	c	131	É	163	£	195	√	227	„
4	<EOT>	36	$	68	D	100	d	132	Ñ	164	§	196	ƒ	228	‰
5	<ENQ>	37	%	69	E	101	e	133	Ö	165	•	197	≈	229	Â
6	<ACK>	38	&	70	F	102	f	134	Ü	166	¶	198	Δ	230	Ê
7	<BEL>	39	'	71	G	103	g	135	á	167	ß	199	«	231	Á
8	<BS>	40	(	72	H	104	h	136	à	168	®	200	»	232	Ë
9	<TAB>	41	)	73	I	105	i	137	â	169	©	201	…	233	È
10	<LF>	42	*	74	J	106	j	138	ä	170	™	202		234	Í
11	<VT>	43	+	75	K	107	k	139	ã	171	´	203	À	235	Î
12	<FF>	44	,	76	L	108	l	140	å	172	¨	204	Ã	236	Ï
13	<CR>	45	-	77	M	109	m	141	ç	173	≠	205	Õ	237	Ì
14	<SO>	46	.	78	N	110	n	142	é	174	Æ	206	Œ	238	Ó
15	<SI>	47	/	79	O	111	o	143	è	175	Ø	207	œ	239	Ô
16	<DLE>	48	0	80	P	112	p	144	ê	176	∞	208	–	240	
17	<DC1>	49	1	81	Q	113	q	145	ë	177	±	209	—	241	Ò
18	<DC2>	50	2	82	R	114	r	146	í	178	≤	210	“	242	Ú
19	<DC3>	51	3	83	S	115	s	147	ì	179	≥	211	”	243	Û
20	<DC4>	52	4	84	T	116	t	148	î	180	¥	212	‘	244	Ù
21	<NAK>	53	5	85	U	117	u	149	ï	181	µ	213	’	245	ı
22	<SYN	54	6	86	V	118	v	150	ñ	182	∂	214	÷	246	ˆ
23	<ETB>	55	7	87	W	119	w	151	ó	183	∑	215	◊	247	˜
24	<CAN>	56	8	88	X	120	x	152	ò	184	∏	216	ÿ	248	¯
25	<EM>	57	9	89	Y	121	y	153	ô	185	π	217	Ÿ	249	˘
26	<SUB>	58	:	90	Z	122	z	154	ö	186	∫	218	/	250	˙
27	<ESC>	59	;	91	[	123	{	155	õ	187	ª	219	€	251	°
28	<FS>	60	<	92	\	124	|	156	ú	188	º	220	‹	252	¸
29	<GS>	61	=	93	]	125	}	157	ù	189	Ω	221	›	253	˝
30	<RS>	62	>	94	^	126	~	158	û	190	æ	222	fi	254	˛
31	<US>	63	?	95	_	127	<DEL>	159	ü	191	ø	223	fl	255	ˇ

2.3 Structures de contrôle

Le déroulé des instructions peut être contrôlé afin de gérer les différents cas de figure, ou des répétitions d'opérations.

2.3.1 Instruction conditionnelle

L'instruction conditionnelle permet de choisir le code à exécuter en fonction de certaines conditions logiques.

```
if condition1 :
    #bloc d'instructions si la condition1 est vraie
elif condition2 :
    #bloc d'instructions si la condition1 n'est pas vraie, mais si condition 2 est vraie
else:
    #bloc d'instructions si aucune condition n'est vraie
```

2.3.2 Boucles inconditionnelles

Lorsqu'on souhaite répéter n fois un bloc d'instructions (et que n est connu), on utilise une boucle inconditionnelle. Celle-ci est introduite sous sa forme la plus intuitive par la syntaxe suivante :

for i **in range(n) :**
    bloc d'instructions

Lors du parcours de cette boucle, i est appelé un compteur. Il va parcourir les valeurs de 0 à $n-1$.

Si l'on souhaite que i parcourt les valeurs de 1 à n on utilisera le mot-clé **range(1,n+1))**.

2.3.3 Boucles conditionnelles

Il arrive parfois que l'on souhaite parcourir une boucle mais que l'on ne sache pas à l'avance combien de fois elle devra être parcourue. Supposons par exemple que l'on souhaite trouver le plus petit diviseur d'un nombre entier naturel donné. Dès lors que celui-ci est trouvé, on souhaite arrêter la boucle (et on a la certitude que ceci arrivera). Ainsi l'arrêt de la boucle se fera au moyen d'une condition.

Ce type de boucles est appelé boucle conditionnelle. Elle est introduite en Python par la syntaxe suivante :

while condition vérifiée :
    bloc d'instructions

2.3.4 Méthodologie d'écritures des boucles

Il est conseillé de suivre une méthode générale pour écrire des boucles. Ainsi :

- Pour écrire une boucle **inconditionnelle** :
    1. On détermine combien de fois la boucle devra s'exécuter

2. On choisit une variable pour le compteur et on détermine si celle-ci joue un rôle dans la boucle
3. On écrit le corps de la boucle
4. On vérifie si on ne doit pas initialiser certaines valeurs avant l'entrée de la boucle et si il ne faut post-traiter des résultats

- Pour écrire une boucle **conditionnelle** :
  1. On identifie la condition pour laquelle la boucle doit être exécutée. Il est souvent plus simple d'exprimer la condition de sortie de la boucle. On peut alors écrire la négation de celle-ci ou plus simplement utiliser l'opérateur booléen not
  2. On écrit le corps de la boucle en s'assurant que celle-ci modifiera effectivement la valeur de la condition à certaines itérations.
  3. On prévoit une initialisation des variables en amont de la boucle
  4. On prévoit éventuellement un post-traitement des variables en sortie de boucle
- Imbriquer des boucles est judicieux lorsqu'on a repéré des procédures doublement (ou triplement ...) répétitives. Un certain nombre de précautions sont à prendre. Pour écrire des **boucles imbriquées** :
  1. On repère quelle variable doit être dans la boucle interne et laquelle dans la boucle externe. En particulier, si l'exécution d'une instruction dépend d'une variable qui dépend elle-même d'une autre variable c'est la première qui doit être placée en boucle interne
  2. On s'interroge si le compteur de la boucle interne doit dépendre ou non de celui de la boucle externe
  3. On écrit le corps de la boucle interne, et ses éventuelles initialisations
  4. On écrit le corps de la boucle externe et ses éventuelles initialisations. En général cette étape est courte.
  5. On vérifie les niveaux d'indentation

3 Analyse des programmes

Pour résoudre un problème, un programme peut rapidement nécessiter un grand nombre d'opérations. Il est alors nécessaire de disposer d'outils pour vérifier la qualité du programme : le variant de boucle pour vérifier la terminaison d'un programme, l'invariant de boucle pour vérifier la correction d'un programme.

3.1 Problème de terminaison des boucles conditionnelles

Un algorithme est un programme de calcul qui s'exécute en un nombre fini d'étapes. Si dans le cadre des boucles inconditionnelles on a la certitude que le calcul se finira effectivement, lors de l'exécution d'une boucle conditionnelle, nous n'avons pas cette certitude.

Il existe une technique pour démontrer qu'une boucle (conditionnelle) s'arrête effectivement : on a recours à un variant de boucle.

Un **variant de boucle** est une quantité entière :

- qui est strictement positive avant l'exécution de la boucle ;
- qui décroît strictement à chaque itération ;
- qui lorsqu'elle est inférieure à un certain nombre rend la condition d'exécution de la boucle conditionnelle fausse.

3.2 Invariant de boucle

Lorsqu'on écrit un programme il est parfois très important de vérifier qu'il est correct, c'est à dire qu'il calcule bien ce qu'on attend. C'est même vital lorsque dans certaines applications industrielles une erreur aurait un impact sur des vies humaines (on pensera par exemple aux programmes qui contrôlent la conduite des lignes de métro automatiques, ou à des programmes touchant au secteur militaire).

Si l'on souhaite montrer qu'une boucle est correcte, on peut tester quelques cas particuliers significatifs, mais ceci ne suffira pas en tout état de cause à montrer que notre boucle est correcte. L'idée est donc d'introduire un outil théorique, appelé invariant de boucle, qui permettra ceci.

Un invariant de boucle est une propriété qui est vérifiée tout au long de l'exécution d'une boucle. Il est ainsi à rapprocher du raisonnement par récurrence en mathématiques. Plus précisément :

Un **invariant de boucle** est une propriété qui :

- est vérifiée avant l'entrée d'une boucle ;
- si elle est vérifiée avant une itération est vérifiée après celle-ci ;
- lorsqu'elle est vérifiée en sortie de boucle permet d'en déduire que le programme est correct.

3.3 Complexité d'un programme

La complexité d'un programme est un critère d'analyse de performance d'un programme qui quantifie les ressources matérielles nécessaires à un programme. Dans la majorité des cas, on se limite à l'estimation d'un ordre de grandeur de ces ressources, en fonction de n, le nombre de données à traiter.

Ordre de grandeur	Nom courant	Temps pour n=50	Temps pour n=10^6	Exemple
$O(1)$	Temps constant	10 ns	10 ns	Accès à un tableau
$O(\log n)$	Logarithmique	20 ns	60 ns	Dichotomie
$O(n)$	Linéaire	500 ns	10 ms	Parcours d'une liste
$O(n \log n)$	Linéarithmique	1 μs	60 ms	Tri par fusion
$O(n^2)$	Quadratique	25 μs	3h	Parcours Tableaux 2d
$O(n^3)$	Cubique	1.25 ms	300 ans	Multiplication Matrices naïve
$O(2^n)$	Exponentielle	135 jours	¿ 10^{300000} années	Problème du Sac à dos
$O(n!)$	Factorielle	10^{48} ans	...	Voyageur de Commerce naïf

4 Manipulation des types structurés

On désigne par « types structurés » les types d'objets informatiques contenant plusieurs éléments de type plus simple ou de type de base. On peut les rencontrer sous d'autres appelations (collections, itérables, etc.). Parmi ceux-ci, nous n'en utiliserons que cinq : les listes `list`, les chaînes de caractères `str`, les n-uplet `tuple`, les dictionnaires `dict` et les tableaux `array`.

type	délimiteur	séparateur	mutable immuable	opérations	accès extraction
Liste					
Tuple					
Chaîne					
Dictionnaire					
Array					

4.1 Copie de listes : pourquoi et comment

4.1.1 Références partagées : mise en évidence

Pour chaque ligne de code ci-dessous, indiquer ce qui est affiché :

```
a = [1, 2]; b = a; a[0] = 0 ; print(b)



a = [1, 2]; b = a[:]; a[0] = 0 ; print(b)



a = [1, [2,3]]; b = a[:]; a[0] = 0; a[1][0] = 0 ; print(b)



.
```

4.1.2 Conséquences

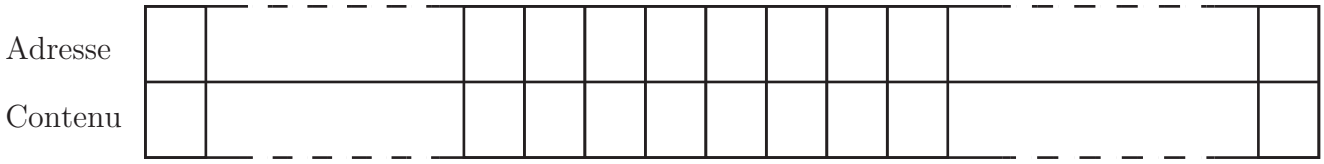
- Pour séparer les références de 2 listes lors d'une affectation `A=B`, il faut faire une copie de la liste `B` : `A=B.copy()` ou `A=B[:]` ;
- Dans le cas de liste de listes, il faut faire des copies profondes en utilisant le module `copy` :

```
import copy
A=copy.deepcopy(B)
```

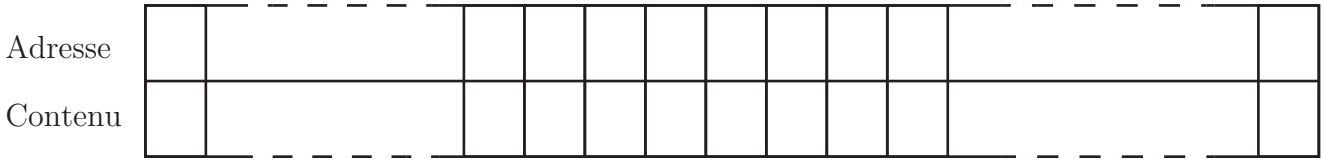
5 Codage des types structurés en RAM

5.1 Cas d'un `float`

En python, les nombres de type float sont codés en double précsion, c'est à dire sur 64 bits ou 8 octets.

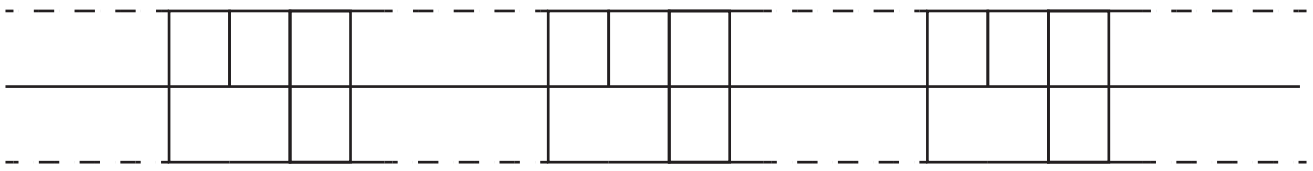


5.2 Cas d'un tableau contigu



- les informations sont organisées en un lieu délimité et fixe de la mémoire ;
- la taille en mémoire est fixe ;
- le contenu ne peut avoir qu'un unique type (tableau de `bool`, ou de `float` par exemple) ;
- le contenu peut changer, pas la taille.

5.3 Cas d'une liste chainée



- les informations sont parsemée dans la mémoire ;
- à la fin du slot du i^{eme} élément, est précisé l'adresse du $i+1^{eme}$ élément ;
- à la fin du dernier élément, aucun successeur n'est précisé.

5.4 Application

5.4.1 Sur les listes chainées

1. Comment accéder au i^{eme} élément d'une liste chainée ?
2. Comment supprimer le i^{eme} élément ?
3. Comment insérer un élément entre le i^{eme} et le $i + 1^{eme}$ élément ?
4. Comment est réalisée la concaténation ?
5. Quelle est la différence entre les deux instructions :

```
s.append(x)
s = s + [x]
```

5.4.2 Comparaison une liste chainée et un tableau contigu

1. Peut-on facilement modifier la longueur (le nombre d'éléments) d'une liste ? Comment ?
2. Peut-on facilement modifier la longueur d'un tableau ?
3. Pourquoi l'addition terme à terme de deux tableaux contigus est-elle plus rapide qu'avec les listes chainée ?
4. Pour réaliser un tri fusion, il vaut mieux utiliser des listes chainées ou des tableaux contigus ?

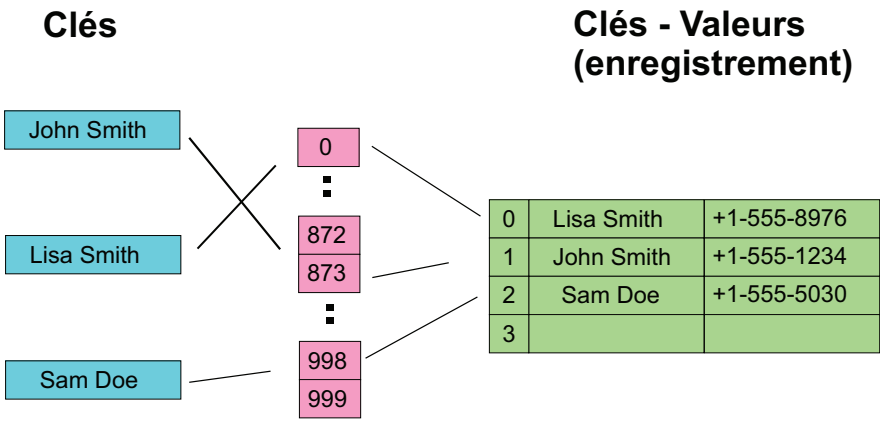
5.5 Cas des dictionnaires

5.5.1 Rappel

0	2001
1	'odyssée'
2	'espace'
3	'Thomas'
4	'Pesquet'

Jean	0612345678
Josephine	0623456789
Jamel	0634567890
Jacques	0645678901
Jimmy	0656789012

5.5.2 Principe



5.5.3 A retenir

- Un dictionnaire est codé en machine en tant que tableau contigu.
- La position d'un enregistrement dans le dictionnaire dépend de sa clé.
- Une fonction de hachage est utilisée pour déterminer l'index de l'enregistrement en fonction de sa clé.
- Cette fonction est utilisée lors de la construction du dictionnaire, et lors de l'accès à un enregistrement du dictionnaire.
- Des procédures spécifiques sont mises en œuvre en cas de collision de deux enregistrements.
- Fonction de hachage :

5.5.4 Remarque : complexité

Recherche dans une liste : $O(n)$
Recherche dans une liste triée : $O(\log n)$
Recherche parmi les clés d'un dictionnaire : $O(1)$