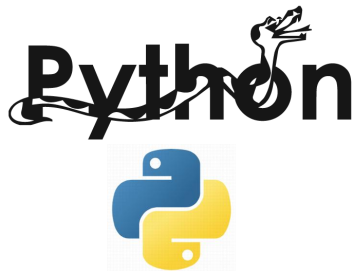


Cours0: Rappel



AU: 2022/2023

1

Les opérations élémentaires: Calculs en algorithmique et **en python**

Opérations	En Algo	En python	Exemple
addition	+	+	2+5 donne 7
soustraction	-	-	8-2 donne 6
multiplication	*	*	6 * 7 donne 42
Exponentiation (puissance)	fonction puissance	**	5 ** 3 donne 125
Division	/	/	7/2 donne 3.5
Reste de la division entière	mod	%	7 % 3 donne 1
Quotient de la division entière	Div	//	7 // 3 donne 2
booléenne	Vrai/ Faux	True/False (Majuscule)	0>1 donne False true donne error
Négation	Non	Not	Not 0==0 donne False
Inégalité	<>	!=	1 != 4 donne True

3

Les premiers pas en python



2

Priorité des opérateurs

- Sous Python, les règles de priorité sont les mêmes que celles qui vous ont été enseignées au cours de mathématique (**PEMDAS**):
 - **P** pour parenthèses. Ce sont elles qui ont la plus haute priorité. Elles vous permettent donc de « forcer » l'évaluation d'une expression dans l'ordre que vous voulez.
Exemple: $2*(3-1) = 4$, et $(1+1)**(5-2) = 8$.
 - **E** pour exposants. Les exposants sont évalués ensuite, avant les autres opérations.
Exemple: $2**1+1 = 3$ (et non 4), et $3*1**10 = 3$ (et non 59049 !).
 - **M** et **D** pour multiplication et division, qui ont la même priorité. Elles sont évaluées avant l'addition **A** et la soustraction **S**, lesquelles sont donc effectuées en dernier lieu.
Exemple: $2*3-1 = 5$ (plutôt que 4), et $2/3-1 = -1$.

4

Les opérations élémentaires: Affectation **en python**

- Une affectation se fait en utilisant le signe =.
- Tester si une égalité est vraie ou fausse se fait en utilisant ==

```
1 >>> a = 13
2 >>> print(a)
3 13
4 >>> a == 13
5 True
```

- Affectation multiple:

```
>>> a, b = 4, 8.33
>>> a
4
>>> b
8.33
```

5

Les opérations élémentaires: Entrées/Sorties **en python**

Avec Python, la fonction **input()** renvoie systématiquement une chaîne de caractères. Donc si c'est un nombre que l'on veut manipuler, il faudra utiliser la fonction **int()** s'il s'agit d'un entier ou **float()** s'il s'agit d'un réel. Par exemple :

```
chaîne = input()
nb = float(chaîne)
```

Ou bien directement

```
nb = float(input())
```

La fonction **print()** affiche le résultat de l'opération ou un message

```
print(4.6 / 2)
```

```
>>> a = "a"
>>> print(a)
a
```

6

Les structures conditionnelles **en python**

```
if test 1 : # un test est une valeur booléenne (logique)
    | bloc si test1 vérifié
elif test 2 : # (facultatif). Sinon si :
    | bloc si test 1 non vérifié mais test2 vérifié
.....
else : # (facultatif). Sinon
    | bloc si aucun des tests précédent n'est vérifié
    suite du programme
```

Exemple

Indentation

```
if x>0:
    print(x,'est positif')
elif x< 0:
    print(x,'est négatif')
else:
    print(x,'est null')
```

```
>>> x=2
```

```
2 est positif
```

7

8

Premiers scripts en Python
Les structures conditionnelles et
itératives

Les structures itératives: Pour **en python**

For variable **in** itérable:
| Bloc répété pour chaque valeur de la variable parcourant l'itérable
suite du programme

```
>>> for i in range(5): # pour i allant de 0 à 4
...     print("blabla")
... 
```

```
blabla
blabla
blabla
blabla
blabla
```

```
>>> for i in range(1, 8): # pour i allant de 1 à 7
...     print(i**2)
... 
```

```
1
4
9
16
25
36
49
```

9

Les structures itératives: Tant que **en python**

While test : # Tant que ...
| Bloc répété tant que test vérifié
suite du programme

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

```
1
2
3
4
5
```

10

Contrôle des boucles

Interrompre une boucle : Break

Le mot clé Break interrompt une boucle. On sort immédiatement de la boucle **for** ou **while** en cours.

```
for x in range(1, 11):
    if x == 5:
        break
    print(x)
print("Boucle interrompue pour x =", x)
```

Court-circuiter une boucle : continue

Avec le mot clé continue, on passe immédiatement à l'itération suivante de la boucle **for** ou **while** en cours. On reprend à la ligne de l'en-tête de la boucle .

11

Les structures de données en Python:
Les chaines, les listes, Les tuples, les
ensembles, les dictionnaires

12

Chaine de caractère :str

Sous Python, une donnée de type string est une suite de caractères délimitée soit par :

```
>>> phrase1 = 'les œufs durs.'
>>> phrase2 = "j'aime bien "
>>> phrase3 = '"N\'est-ce pas ?" répondit-elle.'
>>> phrase4 = """
... C'est plus facile sans l'antislash !
... On peut écrire des antislashes : \
"""
>>> print(phrase1, phrase2, phrase3, phrase4)
```

Simple quote

Double quote

Triple quote

Les chaines de caractères sont ordonnées et non mutables

```
>>> ch = "Mohamed Amin"
>>> print(ch[0], ch[5])
>>> ch[2]='A'
```

'M', 'e'

Erreur

13

Les tuples :tuple

Les tuples correspondent aux listes à la différence qu'ils sont non modifiables. Pratiquement, ils utilisent les parenthèses au lieu des crochets. Ce type de donnée possède les mêmes méthodes que la liste, à l'exception de celles permettant une modification.

```
>>> x = (1,2,3)
>>> x = 1,2,3
>>> len(x)
>>> x[0:2]
>>> x + (3,)
```

Les tuples sont ordonnées et non mutables

```
>>> x[2]
>>> x[2] = 15
```

3

Erreur

15

Les listes :List

Une liste est une structure de données qui contient une série de valeurs et/ou chaînes de caractères, séparés par des virgules et limités par des crochets.

```
>>> animaux = ['girafe','tigre','singé','souris']
>>> tailles = [5, 2.5, 1.75, 0.15]
>>> mixte = ['girafe', 5, 'souris', 0.15]
```

Les listes sont ordonnées et mutables

liste	:	['girafe', 'tigre', 'singé', 'souris']			
indice positif :	0	1	2	3	
indice négatif :	-4	-3	-2	-1	

```
>>> animaux = ['girafe','tigre','singé','souris']
>>> animaux[1]
>>> animaux[0]='lion'
>>> animaux
```

'tigre'

['lion','tigre','singé','souris']

14

Les ensembles : set

Le set est un type Python permettant de créer des collections, formées par des éléments **non dupliqués**.

Son grand intérêt réside dans le fait qu'on peut appliquer à ces objets des opérations propres aux ensembles (union, intersection, différence...), d'où le nom de sets (ensembles).

On crée un set avec la fonction **set**(iterable) ou en énumérant les éléments entre accolades {elem1, elem2...}

```
>>> set1 = {'a', 'b', 'a', 'c'}
>>> type(set1)
```

Les ensembles sont non ordonnées et mutables

```
>>> set1['3']
>>> set1.add('d')
>>> set1
```

Erreur : set not support indexing

{'b', 'c', 'd', 'a'}

16

Les dictionnaires: Dict

Les dictionnaires se révèlent très pratiques lorsque vous devez manipuler des structures complexes à décrire et que les listes présentent leurs limites.

- on crée un dictionnaire vide avec les symboles {} ou avec la fonction `dict()`.
- Ensuite, on remplit le dictionnaire avec différentes clés auxquelles on affecte des valeurs (une par clé).
- Pour récupérer la valeur d'une clé donnée, il suffit d'utiliser une syntaxe du style `dictionnaire['cle']`.

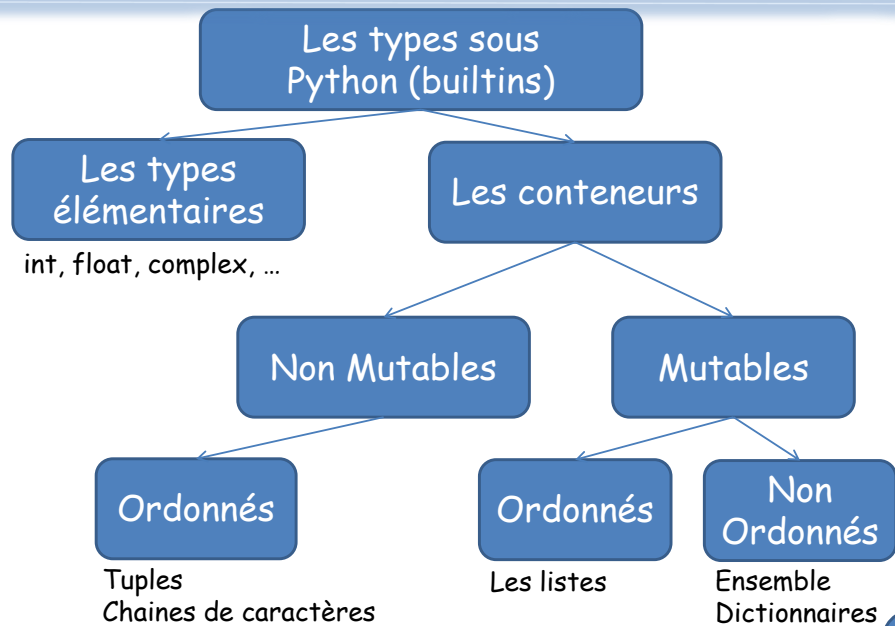
```
>>> ani1 = {}
>>> ani1['nom'] = 'girafe'
>>> ani1['taille'] = 7.0
>>> ani1['poids'] = 1200
>>> ani1
```

Les dictionnaires sont non ordonnés et mutables

```
>>> ani1['taille'] → 7.0
>>> ani1['taille']=11.0
>>> ani1 → {nom:'girafe, taille:11.0, poids:1200 }
```

17

Les types sous Python



19

Compréhension de liste

- Pour construire une liste à l'aide d'une boucle **for** assortie éventuellement d'une condition
Exemple:

```
# L'expression ci-dessous fabrique la liste des carrés des  
# nombres pairs de 1 à 10  
carres = [ nb*nb for nb in range(1,11) if nb%2==0 ]
```

Équivalent à

```
carres = []
for nb in range(1,11):
    if nb%2 == 0:
        carres.append(nb*nb)
```

- Pour construire une liste à l'aide d'une boucle **for** assortie éventuellement d'une expression conditionnelle

```
# L'expression ci-dessous parcourt les entiers de 1 à 9 et affiche les valeurs impaires, sinon 0  
[ x if x%2!=0 else 0 for x in range(1,10) ]
```

18

Les fonctions en python

20

Les fonctions en python

Définir une fonction

```
def nomFonction( parametres ):  
    # instruction 1 de la fonction  
    # instruction 2 de la fonction  
    # ...  
    # instruction n de la fonction
```

- ☐ Lors de la définition d'une fonction, le code n'est pas exécuté;
- ☐ A tout moment dans le programme ou dans une fonction on peut appeler la fonction avec la commande:

Appeler une fonction

```
nomFonction( parametres )
```

Fonction Lambda

Pour les fonctions donc le corps ne contiendrait qu'une seule instruction return expression, Python offre la possibilité de définir une fonction anonyme aussi appelée lambda fonction (ou lambda expression). Sa syntaxe est :

lambda arguments... : expression

```
# La lambda fonction ci-dessous :  
somProd = lambda n1, n2: (n1+n2, n1*n2)
```

Équivalent à

```
def somProd(n1, n2):  
    return (n1+n2, n1*n2)
```

Dans les 2 cas :

```
somProd(3, 10)
```

Les fonctions en python

```
>>> def hi(name):  
...     print("hello " + name + " from Python!!!")  
...  
>>> hi("John")  
hello John from Python!!!
```

```
>>> def f(x):  
...     return 2 * x + 1  
...  
>>> # on peut maintenant l'utiliser  
>>> f(0)  
1  
>>> # et la réutiliser  
>>> f(5)  
11
```

Fonctions prédéfinies de Python

▪ Certaines fonctions résident dans des **modules** spécialisés, comme fractions, math, random, os, time, calendar, ...

Exemples :

Calculer le pgcd:

```
>>> import fractions  
>>> fractions.gcd ( 18,12)  
6  
>>> from fractions import gcd  
>>> gcd ( 18,12)  
6
```

Générer des entiers au hasard :

```
>>> import random as rd  
>>> rd.randint ( 1,10)  
9  
>>> rd.randint ( 1,10)  
6  
>>> rd.randint ( 1,10)  
10
```

Les modules

- Les modules sont des programmes Python qui contiennent des fonctions que l'on est amené à réutiliser souvent (on les appelle aussi bibliothèques ou libraires).

■ **import <nom du module> :**

import random

random.randint(0,10)

L'instruction import permet d'accéder à toutes les fonctions du module random. Ensuite, nous utilisons la fonction (ou méthode) randint(a,b) du module random

- import random as rand**
rand.randint(1, 10)

il est également possible de définir un alias (un nom plus court) pour un module

■ **from <module> import <fonction> :**

- from random import randint**
randint(0,10)

À l'aide du mot-clé **from**, on importe une fonction spécifique d'un module donné.

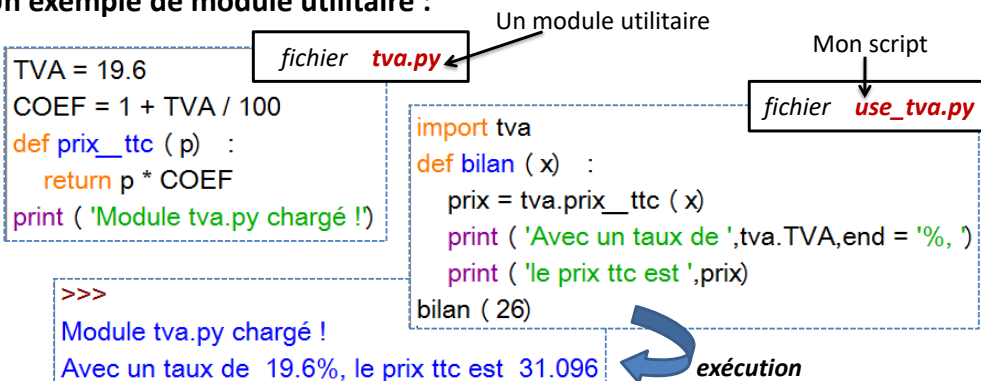
- from random import ***

On peut également importer toutes les fonctions d'un module

Création de modules

- Un **module** est un fichier **xxxxxx.py** écrit en Python et contenant :
 - des définitions
 - des instructions (par exemple d'affichage)
- Un module peut être destiné à être utilisé par un autre module. Il exporte alors un certain nombre de fonctionnalités.

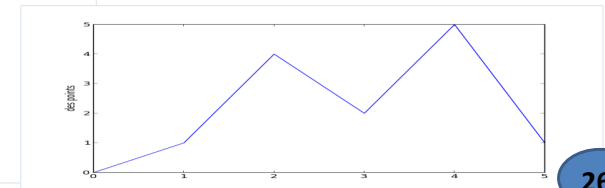
Un exemple de module utilitaire :



Exemple de modules Python

- La bibliothèque **numpy** propose un ensemble de modules, de classes, d'objets et de fonctions dédiés aux calculs numériques.
- La bibliothèque **Scipy** basée sur les tableaux numpy présente un grand choix de modules pour le calcul scientifique.
- La bibliothèque **Matplotlib** est très riche pour le traçage de courbes et de graphiques.

```
import matplotlib.pyplot as plt
l=[0,1,4,2,5,1]
plt.plot(l)
plt.ylabel('des points')
plt.show()
```



26

La récursivité

La récursivité

- ❑ En mathématique, la récurrence
- ❑ Récursive: Toute fonction ou procédure qui fait un appel à elle-même.

Exemple:

La fonction factorielle vérifie : $0!=1$ et $(n+1)!=(n+1)\times n!$

En algorithmique

```
Fonction fact(n:entier) :entier
r:entier
Si (n=0 ) Alors
  r ← 1
sinon
  r ← n * fact(n-1)
Retourner(r)
```

En python

```
def fact(n):
  if n==0:
    r=1
  else:
    r = n * fact(n - 1)
  return r
```

Structure générale de la fonction Récursive en python

```
def fonction_réursive (paramètre):
  if ... :                # close d'arrêt
    return ...            # cette ligne détermine le retour de la
                           # variable de sortie dans le cas de base
  elif .... :             # cette ligne est ajoutée que lorsqu'on
                           # a une autre close d'arrêt
    return ...
  else :
    return ...            # instructions contenant un appel à la
                           # fonction_réursive
```

Principe d'une Fonction récursive

Fonction récursive Toute fonction ou procédure qui fait un appel à elle-même.

■ Deux propriétés sont à respecter pour s'assurer du bon fonctionnement d'une fonction/procédure récursive :

- elle doit contenir un cas de base (**condition d'arrêt**)
- elle doit s'appeler elle-même, en modifiant son état, pour pouvoir se ramener au cas de base.

■ Dans l'exemple de la fonction **fact** :

- Un cas de base $\text{fact}(0)=1$
- Un appel à $\text{fact}(n-1)$ qui va de proche en proche nous ramener à $\text{fact}(0)$.

```
def fact ( n ) :
  if n == 0:
    return 1
  else:
    return n * fact ( n-1)
```

Gestion d'erreur

Gestion d'erreur

- En Python, on utilise la construction **try: ... except: ...**

```
Try:
    <instr1>
...
Except:
    <instr2>
...
```

□ Si au cours de l'exécution du bloc d'instructions <instr 1 >... une exception se produit, l'exécution du bloc est abandonnée et le bloc <instr 2 >... est exécuté.

□ Si l'exécution du bloc <instr 1 >... s'est déroulée normalement, le bloc **except** n'est pas utilisé.

33 3
3

Gestion d'erreur

- Exemple:

```
while 1:
    try:
        nb = int( input( "Entrez un nombre: ") )
        print( "Le nombre est", nb )
        break
    except:
        print ( "Vous n'avez pas entré un nombre !" )
        print ( "Essayez encore" )
```

Entrez un nombre: aabb

- les exceptions permettent d'exprimer le comportement général d'un calcul tout en sachant qu'il pourra y avoir quelques cas d'erreurs exceptionnelles qui seront traités à part.

- Il y a différentes exceptions : **ValueError**, **ZeroDivisionError**, **KeyError**, etc. Un bloc **except** les attrape toutes.

34 3
4

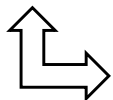
Gestion d'erreur

- L'instruction **raise** permet de lever volontairement une exception. On peut trouver l'instruction à tout endroit du code, pas seulement dans un bloc try

- l'instruction **assert** permet de déclencher une exception si la condition est fausse : `assert (<condition>)`

```
def KelvinToFahrenheit ( Temperature ) :
    try:
        assert ( Temperature >= 0 )
        return ( ( Temperature-273 ) *1.8 ) +32
    except :
        print ( 'erreur : temperature négatif '
```

équivalent



```
def KelvinToFahrenheit ( Temperature ) :
    try:
        if Temperature < 0:
            raise
        return ( ( Temperature-273 ) *1.8 ) +32
    except :
        print ( 'erreur : temperature négatif '
```

Manipulation des fichiers

Manipulation des fichiers

- ❑ Ouverture en lecture : `f_in = open('foo.txt','r')`
- ❑ Ouverture en écriture : `f_out = open('foo.txt','w/a')`
 - l'option '+' : le fichier est ouvert en lecture et en écriture.
 - l'option 'b' : ouverture d'un fichier binaire.
- ❑ Fermeture :
`f.close()`
- ❑ Lecture :
`f_in.read()` : lit l'ensemble du fichier et le renvoie sous forme de chaîne.
`f_in.readline()` : lit et renvoie une ligne du fichier de `f`,
la fin de ligne (`\n`) est incluse
`f_in.readlines()` : lit et renvoie une liste de toutes les lignes du fichier de `f`,
où chaque ligne est représentée par une chaîne se terminant par `\n`
- ❑ Ecriture :
`f_out.write(s)` : écrit la chaîne `s` dans le fichier de `f`
`f_out.writelines(lst)` : écrit la liste de chaîne `lst` dans le fichier de `f`

La complexité

38

Complexité

Définitions :

Afin de comparer plusieurs algorithmes résolvant un même problème, on introduit des mesures de ces algorithmes calculant leurs coûts, appelées **complexité**.

➡ Chercher l'algorithme dont la complexité est minimale.

❑ **Complexité temporelle** : c'est le nombre d'opérations élémentaires effectuées par une machine qui exécute l'algorithme.

❑ **Complexité spatiale** : c'est le nombre de positions mémoire utilisées par une machine qui exécute l'algorithme.

Complexité

Propriétés :

Ce que nous intéresse:

- ❑ Ressources : nombres d'instructions (complexité temporelles)
- ❑ En fonction du paramètre **n**, Complexité = **C(n)**
- ❑ Asymptotique $\uparrow n$: ordre de grandeur **O(n)**

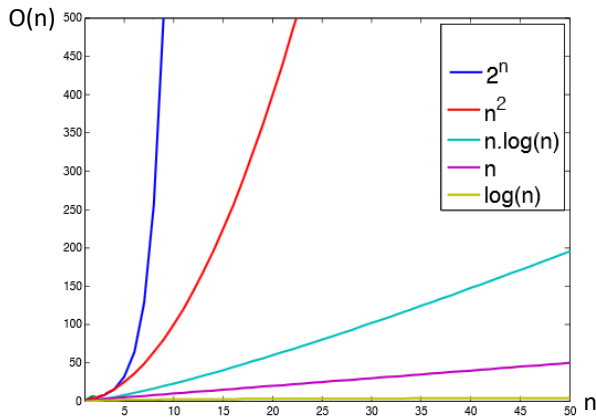
$$C(n) = n^2 + bn + c = O(n^2)$$

$$C(n) = 2^n + bn^2 + c = O(2^n)$$

- ❑ Pire des cas

Algorithme:
Entrée = **n**

Comparaison :



La complexité $O(n)$ en fonction de n

Quelques règles pour les algorithmes itératifs :

- Affectation, opérations arithmétiques: $O(1)$
- Instruction Si -Sinon : O (complexité max des deux branches)
- Séquence d'opérations : l'opération la plus coûteuse domine (règle de la somme)

4
1

Principaux types de données en python

Type	Description	Exemple
int	Entiers 32 bits	3, -4
long	Entiers > 32 bits	101L
float	Nombre à virgule	3.0, 6.55
complex	Nombre complexe	1+2j
bool	booléen	True, False
str	Chaîne de caractère	'az', "wafa"
tuple	Séquence	(1,5,9)
list	Séquence	[1,3,'a']
dict	dictionnaire	{1:'x',2:'y'}
set	ensemble	{'a','b','a'}

43

Annexe

Principaux fonctions en python

Fonctions	Sortie
abs(x)	valeur absolue de x
dict(x)	dictionnaire vide
float(x)	convertit int ou string en float
id(obj)	adresse mémoire de obj
int(x)	convertit float ou string en int
len(s)	nombre d'objets dans la séquence s
list()	liste vide
max(s), min(s)	valeur maximale, minimale dans la séquence s
pow(x,y)	$x^{**}y$
range(x)	liste de x entiers allant de 0 à x-1
str(obj)	convertit obj en chaîne de caractères
sum(s)	somme de la séquence numérique s
tuple(items)	convertit les items en tuple
type(obj)	donne le type de obj

44

Les opérations sur les chaînes de caractères (1)

Fonctions et méthodes des chaînes	sa, sb sont des objets chaînes (str)
<code>sa=chr(n),</code>	Renvoie un caractère dont on connaît le code ASCII n
<code>n=ord(sa)</code>	Renvoie le code ASCII n d'un caractère sa.
<code>sa.count(sb, *deb, *fin)</code>	Compte le nombre de fois où sb est dans sa entre les indices de b et fin.
<code>sa.find(sb, *deb, *fin)</code>	Retourne l'indice de la première apparition de sb dans sa entre les index deb et fin.
<code>sa.replace(st_anc, st_nouv, *n)</code>	Remplace les n premières apparitions de st_anc par st_nouv.
<code>sa.split(sep, *n)</code>	Sépare sa en n coupures maximum au niveau de chaque chaîne sep.

45

Les opérations sur les listes

Fonctions et méthodes des listes	Liste vide : <code>L = []</code> , numérotée à partir de 0 et de longueur <code>len(L)</code> .
<code>L.append(élément)</code>	Ajoute élément à la liste L. Peut aussi s'écrire <code>L = L + [élément]</code> .
<code>L.insert(index, élément)</code>	Insère élément à la position index.
<code>L.extend(K)</code>	Ajoute tous les éléments de la liste K à la liste L. On peut aussi utiliser les opérateurs + et *
<code>L.remove(élément)</code> <code>L.pop(*index)</code>	Supprime de L ; erreur si élément n'est pas présent. Supprime l'élément n° index; si index n'est pas indiqué, le dernier élément est enlevé.
<code>L.index(élément)</code>	Indique la position de élément; si pas présent, une erreur se produit.
<code>L.sort()</code> <code>M = sorted(L)</code>	Tri de L. Retourne la liste L triée dans M.

47

Les opérations sur les chaînes de caractères (2)

Fonctions et méthodes des chaînes	sa, sb sont des objets chaînes (str)
<code>sa.strip(*s)</code>	Supprime tous les espaces au début et à la fin de sa. Si s est donné, tous les caractères faisant partie de s sont supprimés de sa.
<code>sa.upper()</code> <code>sa.lower()</code>	Change la casse minuscules vers majuscules ou majuscules vers minuscules de sa.
<code>sa.capitalize()</code>	Transforme en majuscule le premier caractère de la chaîne sa.
<code>sa.join(Liste_mots)</code>	Crée une chaîne à partir des mots de Liste_mots, séparés par sa.
Formatage des nombres	La méthode <code>s.format(...)</code> permet de combler ces trous en donnant la liste des valeurs à mettre dans les trous : <pre>>>> s = "Voici {0} chaîne à {1} trous." >>> print(s) >>> print(s.format("une ", 2))</pre>

Les opérations sur les listes

Fonctions et méthodes des listes	Liste vide : <code>L = []</code> , numérotée à partir de 0 et de longueur <code>len(L)</code> .
<code>élément in L</code> <code>élément not in L</code>	Booléen disant si élément est dans L. Très pratique dans un if pour remplacer une série de or : au lieu de <code>(x==1) or (x==2) or (x==3)</code> on mettra <code>x in [1, 2, 3]</code> .
<code>L[i : j * : k]</code>	Renvoie une tranche (slice) constituée des éléments n° i à j - 1 de L (si k est indiqué le compte ira par pas de k). Peut servir à insérer une autre liste entre i et j : <code>L[i : j]=M</code> .
<code>del L[i : j]</code>	On peut aussi supprimer les éléments avec <code>L[i : j] = []</code> .
<code>min(L), max(L), sum(L)</code>	Min, max, somme des éléments de la liste L.

48

Les opérations sur les tuples

Fonctions et méthodes des tuples	sortie
tuple(itérable)	La fonction tuple fonctionne exactement comme la fonction list, c-à-d qu'elle prend en argument un objet itérable et renvoie le tuple correspondant.
T.count(obj)	retourne le nombre d'occurrences de obj dans T
T.index(obj)	retourne la position de la première occurrence de obj dans T

49

Les opérations sur les dictionnaires

Fonctions et méthodes des dictionnaires	sortie
d.keys()	Renvoie les clés d'un dictionnaire (sous forme de liste)
d.values()	Renvoie les valeurs d'un dictionnaire (sous forme de liste)
d.items()	les clés et les valeurs d'un dictionnaire
d.clear()	Vider le dictionnaire d.
del d[clé]	Supprime l'item ayant la clé clé du dictionnaire.
d.pop(clé)	Renvoie la valeur ayant clé clé et supprime l'item (clé, valeur) correspondant du dictionnaire.

51

Les opérations sur les ensembles

Fonctions et méthodes des ensembles	sortie
set1 = set(itérable) set2 = set()	La fonction set prend en argument un objet itérable et renvoie l'ensemble correspondant.
set2 = {1, 2} set2.add(3)	Ajout dans un ensemble set3.
set1 set2	set 1 union set2
set1 & set2	Intersection entre set1 et set2
set1 - set2	différence
set1 ^ set 2	Différence symétrique
< , <= , > , >=	Relations d'inclusion entre ensembles. {1,2} <= {1,2,3,4} retourne True.

Complexité: technique

Comment calculer la complexité d'un algorithme?

→ La technique : **incrémenter et compter**

Comparer $C(n+1)$ et $C(n)$

❑ Si $C(n+1) \approx C(n)$: Alors Complexité constante: $O(1)$

❑ Si $C(n+1) \approx C(n) + 1$: Alors Complexité linéaire: $O(n)$

❑ Si $C(n+n) \approx C(2n) \approx C(n) + 1$: Alors Complexité logarithmique $O(\log_2(n))$

❑ Si $C(n+1) \approx C(n)+n$ Alors Complexité polynomiale $O(n^2)$

❑ Si $C(n+1) \approx 2 * C(n)$ Alors Complexité exponentielle $O(2^n)$

Algorithme:
Entrée = n

Complexité linéaire:

Algorithme:

Entrée = **n**

Var **n**: entier

Début

Lire(n) (1)

Pour i de 1 à **n** faire (n)
 écrire(i)

Fin pour

Complexité dépend de l'entrée n:

$$C(n) = 1 + n$$

$$C(n+1) = 1 + n + 1 = C(n) + 1$$

→ Complexité est **O(n)**

Exemples:

- ❑ Chercher un élément dans un tableau non ordonné de taille n
- ❑ Calcul de la longueur d'un tableau de taille n

5
3

```
def dichotomie(t, v):
```

```
    a = 0
```

```
    b = len(t) - 1
```

```
    while a <= b:
```

```
        m = (a + b) // 2
```

```
        if t[m] == v:
```

```
            # on a trouvé v
```

```
            return True
```

```
        elif t[m] < v:
```

```
            a = m + 1
```

```
        else: b = m - 1
```

```
        # on a a > b
```

```
        return False
```

Complexité logarithmique:

Complexité:

n	i	m
10	1, 2 (2 instructions)	10,5,2

n	i	m
20 = 10*2	1, 2,3 (3 instructions)	20, 10,5,2

$$C(2n) = 3 + 1 + 1 = 5$$

$$C(n) + 1 = 2 + 1 + 1 + 1 = 5$$

$$C(2n) = C(n) + 1$$

→ Complexité est **O(log₂(n))**

Exemple:

- ❑ Chercher un élément dans un tableau ordonné (recherche dichotomique)

5
4

Manipulation des fichiers:

Fonction	description
f=open(nom_du_fichier,'r')	Ouvre le fichier nom_de_fichier (donné sous la forme d'une chaîne de caractères indiquant son emplacement) en lecture (r comme read). Le fichier doit exister et seule la lecture est autorisée.
f=open(nom_du_fichier,'w')	Ouvre le fichier nom_de_fichier en écriture (w comme write). Si le fichier n'existe pas, il est créé, sinon il est écrasé (vidé avant utilisation).
f=open(nom_du_fichier,'a')	Ouvre le fichier nom_de_fichier en ajout (a comme append). Identique au mode 'w' , sauf que si le fichier existe, il n'est pas écrasé et ce qu'on écrit est ajouté à partir de la fin du fichier.
f.close()	Sur un fichier ouvert comme précédemment, le ferme. Cette ligne est impérative pour les fichiers ouverts en écriture, puisque le fichier n'est réellement écrit complètement qu'à la fermeture.

Manipulation des fichiers:

Fonction	description
<code>f.read()</code>	Lit tout le fichier d'un coup et le renvoie sous forme de chaîne de caractères (à ne réserver qu'aux fichiers de taille raisonnable).
<code>f.readlines()</code>	Pareil que précédemment, mais le résultat est une liste de chaînes de caractères, avec un élément de la liste par ligne. Attention, le saut de ligne <code>\n</code> est présent à la fin de la ligne.
<code>f.readline()</code>	Lit une unique ligne du fichier et la renvoie sous forme de chaîne (avec <code>\n</code> au bout). Le curseur de lecture (virtuel !) est positionné au début de la ligne suivante. Si on est arrivé en fin de fichier, le résultat est une chaîne vide <code>""</code> .
<code>f.write(s)</code>	Écrit la chaîne <code>s</code> à la suite du fichier.
<code>f.writelines(T)</code>	Écrit l'ensemble des éléments de <code>T</code> dans le fichier <code>f</code> comme des lignes successives. T est une liste, une séquence, un tuple... bref, un itérable.

Enfin, un fichier ouvert en lecture peut être vu comme un itérable : on peut parcourir ses lignes avec une boucle `for` sans passer par *readlines* :

```
f=open('nom_du_fichier','r')
```

```
for ligne in f:
```

```
    [instructions] # par exemple print(ligne)
```