

Notions de Base : Rappel

Aide-Mémoire pour la préparation de l'épreuve
d'Informatique du Concours National d'Entrée aux
Ecoles d'Ingénieurs MP-PC-T 2020

Elaboré par :



Houcemeddine Filali (IPEIM)

Laila Hammami (IPEIN)

Houssemeddine Bouzidi (IPEIM)

Wafa Meftah (IPEIS)

Hayet Latrach (IPEIT)

Mohamed Hammami (IPEIEM)

Besma Fayeche (ENSIT)

Préambule

Ce document est destiné aux étudiants de deuxième année MP-PC-T des Instituts Préparatoires aux Etudes d'Ingénieurs.

Dans les circonstances exceptionnelles liées à la crise sanitaire mondiale et nationale du Covid19, un ensemble de supports de révision pour le programme d'informatique du concours national d'entrée aux écoles d'ingénieurs ont été élaborés essentiellement par les membres de la commission informatique du concours, suite à une action bénévole.

Le but de ces supports est d'accompagner les étudiants de tout le territoire Tunisien dans leur révision en leur fournissant à tous les mêmes documents, afin d'assurer une équité entre eux, qu'ils bénéficient ou non d'un enseignement ou tutorat à distance.

Toutefois, ces documents sont considérés comme des aide-mémoires et ne remplacent pas les cours assurés en présentiel.

Le document présent comporte un rappel sur les notions de base associées à Python comme la manipulation des itérables, les structures de contrôle, les fichiers mais également sur la récursivité et la complexité algorithmique.

Table des matières

I	Introduction	1
II	Variable et affectation	1
III	Opérateurs	2
IV	Classes élémentaires	2
V	Opérations d'entrée/sortie	3
VI	Structures de contrôle	4
VII	Classes itérables	5
VII.1	Opérations communes aux itérables	5
VII.2	Les séquences	6
VII.2.1	Opérations communes aux séquences	6
VII.2.2	La classe str	7
VII.2.3	La classe list	8
VII.2.4	La classe tuple	8
VII.3	Les itérables non ordonnés	9
VII.3.1	La classe set	9
VII.3.2	La classe dict	10
VIII	Construction des itérables en compréhension	11
VIII.1	Listes en compréhension	11
VIII.2	Tuples en compréhension	11
VIII.3	Ensembles en compréhension	12
VIII.4	Dictionnaires en compréhension	12
IX	Import de bibliothèques ou modules	12
X	Les sous-programmes	13
XI	Gestion des exceptions	15
XII	Gestion des fichiers	15
XII.1	Fichiers textes	15
XII.2	Fichiers binaires	17
XIII	La récursivité	17
XIV	Complexité algorithmique	18
XIV.1	Calcul du coût	18
XIV.2	Complexité asymptotique	19
XIV.3	Classes de complexité asymptotique	19
	Bibliographie et Netographie	20

I Introduction

Python est un langage de programmation multi-paradigmes (procédural, orienté objet et fonctionnel) et dynamiquement typé. Sa version standard est libre et est disponible sur le site officiel de Python « python.org ».

La version Python3.X est adoptée par le programme officiel du cycle préparatoire. Cette version inclut un grand nombre de modules pré-programmés fournis d'office, tels que `math`, `cmath`, `random`, etc., mais n'inclut pas les modules tels que `numpy`, `scipy` et `matplotlib` qu'il faut importer explicitement au besoin. Un module regroupe des ensembles de variables et de fonctions apparentées, que l'on appelle également bibliothèque.

Anaconda est une solution comportant l'interpréteur Python livré avec plusieurs environnements de développement intégrés (IDLE, spyder, Jupyter) et inclut une panoplie de modules (`numpy`, `scipy` et `matplotlib`) mais qui nécessite beaucoup de ressources.

II Variable et affectation

Une variable est une référence associée à un objet déjà créé en mémoire. Elle n'a pas de type et n'est pas déclarée. Toute variable doit être affectée avant son utilisation. L'identificateur (nom) de la variable ne peut contenir que des caractères alphanumériques et le caractère '_' et ne débutant pas par un chiffre.

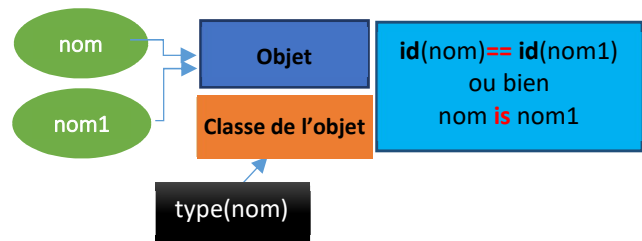
L'affectation permet d'associer un nom à un objet (instance d'une classe) en mémoire :

```
>>> nom = objet
```

Ainsi une affectation de la forme :

```
>>> nom1 = nom
```

réfère le même objet avec autre nom (`nom1`).



On obtient le type d'un objet à l'aide de la fonction `type(objet)` et son adresse mémoire désignant son identité à l'aide de `id(objet)`.

III Opérateurs

Description	Opérateurs	Exemples/Remarques
Affectation	<code>v = valeur</code> <code>v1=v2=v3= valeur</code> <code>v1,v2,v3=valeur1,valeur2,valeur3</code>	<code>x = 5</code> <code>x = y= z = t = 5</code> <code>x, y, z = 1, '2', 5</code>
Parenthèses	<code>()</code>	<code>(x+y)*2</code> #donne 6
Opérations arithmétiques		
Puissance	<code>x**y</code> # ou <code>pow(x, y)</code>	<code>1j**2</code> #donne <code>-1+0j</code>
Signes	<code>+</code> ; <code>-</code>	
Opérateurs multiplicatifs	<code>x*y</code> ; <code>x / y</code> # produit et division réelle de x par y <code>x//y</code> ; <code>x%y</code> #quotient et reste de division entière de x par y	<code>4/2</code> #donne <code>2.0(float)</code> <code>4//2</code> # donne 2 (int) <code>4%2</code> # donne 0
Opérateurs additifs	<code>x+y</code> ; <code>x-y</code>	
Opérateurs relationnels		
Comparaison d'identité	<code>is</code> <code>is not</code>	
Comparaison de valeur	<code>x==y</code> ; <code>x!=y</code>	<code>2!=5</code> # donne True <code>2==5</code> # donne False
Comparaison riche	<code><</code> ; <code>></code> ; <code><=</code> ; <code>>=</code>	
Opérateurs logiques		
Produisent un booléen (bool)	<code>and</code> ; <code>or</code> ; <code>not</code>	
Opérateurs avec assignation		
Produisent le même résultat que l'affectation du résultat de l'opération effectuée	<code>x += y</code> # <code>x = x + y</code> <code>x -= y</code> # <code>x = x - y</code> <code>x *= y</code> # <code>x = x * y</code> <code>x /= y</code> # <code>x = x / y</code> <code>x //= y</code> # <code>x = x // y</code> <code>x %= y</code> # <code>x = x % y</code> <code>x **= y</code> # <code>x = x ** y</code>	

IV Classes élémentaires

Les classes élémentaires sont immuables (non modifiables). Les opérations appliquées aux instances de ces classes génèrent de nouvelles instances comme le montre l'exemple suivant :

```
>>> x=5
>>> id(x)
2053437424
>>> x+=1 # x référence une nouvelle instance
>>> id(x)
2053437440
```

Classes élémentaires	Description	Exemples / Remarques
NoneType	Classe de l'objet None (objet n'ayant pas de valeur)	<code>type(None)</code> # donne NoneType
Bool	Booléens True et False sont les seules instances de cette classe.	<code>type(True)</code> # donne bool <code>bool(20)</code> # donne True <code>bool(0)</code> # donne False <code>x=y=2; x==y</code> # donne True <code>x,y=1,2; x==y</code> # donne False
Int	Entiers relatifs (Z)	<code>y=2 ; type(y)</code> # donne Int <code>x = int('52')</code> # x prend 52 <code>int(22.5)</code> # donne 22
Float	Nombres flottants (IR)	<code>y=3.5 ; type(y)</code> # donne Float <code>x = float('22.2')</code> #x prend 22.2 <code>x = -1e33</code> #x prend -1e+33
Complex	Nombres complexes (C)	<code>z = 3+1j</code> #ou <code>z = complex(3,1)</code> <code>type(z)</code> # donne Complex <code>complex(5)</code> # donne (5+0j) <code>z.real</code> # donne 3.0 (float) <code>z.imag</code> # donne 1.0 (float) <code>z.conjugate()</code> # donne 3-1j

Remarque

`bool (x)` # retourne False si l'évaluation de x est nulle et True autrement

`int(x)` # retourne un entier si x est un nombre ou une chaîne de caractère contenant un nombre entier.

`float(x)` # retourne un réel si x est un nombre ou une chaîne de caractère contenant un nombre réel.

`complex(x,y)` # retourne un nombre complexe formé par la partie réelle x et la partie imaginaire y.

V Opérations d'entrée/sortie

Description	Syntaxe	Exemples
Sortie	<code>print(expr1, ... , exprn)</code>	<code>print('bienvenue à la formation')</code>
Entrée Input prend (optionnel) et retourne une chaîne de caractères	<code>nom = input()</code> <code>nom = input("message")</code>	<pre>>>> nom= input('votre nom svp: ') votre nom svp: Foulen >>> print('Bonjour',nom) Bonjour Foulen</pre>

VI Structures de contrôle

Un bloc de code est un groupe d'instructions Python indentées précédées par ":". Un bloc peut en inclure d'autres.

Description	Syntaxe	Exemples
Structure conditionnelle		
Permet de conditionner l'exécution d'un ou plusieurs blocs d'instructions, mais un seul est exécuté.	<pre> if cond : bloc1 elif cond2 : bloc2 ... else : blocn </pre>	<pre> #maximum de 3 entiers x, y, z if x > y and x > z : print(x) elif y > z: print(y) else : print(z) </pre>
Boucle conditionnelle		
Permet d'exécuter un bloc tant que la condition d'exécution de la boucle est à True. La condition est toujours testée avant l'exécution du bloc de la boucle.	<pre> while cond_exécution : bloc </pre>	<pre> #pgcd de x et y while x % y != 0 : x, y = y, x % y print(y) </pre>
Permet d'exécuter un bloc jusqu'à ce que la condition d'arrêt devienne True. La condition d'arrêt est toujours testée après l'exécution du bloc de la boucle.	<pre> while True : bloc if cond_arrêt : break </pre>	<pre> #saisie d'une voyelle while True: v=input('saisir une voyelle:') if v in ['a','e','i','o','u','y']: break </pre>
Boucle NON conditionnelle		
Permet d'exécuter le bloc pour chaque élément v de l'itérable. Un itérable est une collection d'objets (voir section VII sur les itérables).	<pre> for v in itérable: bloc </pre>	<pre> #calcul du produit des éléments d'une liste p = 1 for v in [1,8,7]: p*=v </pre>
Expression conditionnelle		
Python offre la possibilité de former des expressions dont l'évaluation est soumise à une condition. Le résultat est l'évaluation de expression1 si la condition est True, et celle de expression2 sinon. Cette construction <u>ne permet pas</u> l'utilisation du mot réservé elif mais on peut emboîter deux expressions conditionnelles.	<pre> expr1 if cond1 else expr2 expr1 if cond1 else expr2 if cond2 else expr3 </pre>	<pre> #maximum de 2 valeurs x ,y m = x if x > y else y #les éléments négatifs de L1 sont remplacés par 0 dans L2 (voir l'itérable liste dans la section VII) L1=[-1,0,5,3] L2=[0 if x<0 else x for x in L] x=int(input("?")) ">0" if x>0 else "<0" if x<0 else "=0" </pre>

VII Classes itérables

Les classes itérables ou conteneurs permettent de conserver une collection d'objets. Ces classes peuvent se ramifier selon deux critères :

- La mutabilité : classes modifiables (list, set, dict) et classes non modifiables (range, str et tuple).
- L'ordre : classes ordonnées appelées encore séquences (str, tuple, list, range) et classes non ordonnées (set et dict).

VII.1 Opérations communes aux itérables

Le tableau suivant présente quelques opérations communes applicables aux instances des itérables.

Description	Syntaxe	Exemples / Remarques
Taille d'un itérable	len (iter)	lst = [1,2,5] len(lst) # donne 3
Test d'appartenance d'un élément x à un itérable	x in iter x not in iter	5 in (1,5,[1,3]) # donne True 3 not in (1,5,[1,3]) # donne True "li b" in "ali baba" # donne True "Ali" in "ali baba" # donne False
Somme des éléments d'un itérable (itérable de valeurs numériques uniquement).	sum (iter)	sum([1,5,2,10]) # donne 18
Valeur maximale/ minimale TypeError pour un itérable comportant des objets non comparables	max (iter) min (iter)	max([1,5,2,10]) # donne 10 l = (1+3j,2+5j,11j); max(l) # Error min(1,5,2,10) # donne 1
Tri d'un itérable d'objets comparables (TypeError si l'itérable comporte un objet non comparable) Renvoie une liste contenant les éléments dans l'ordre.	sorted (iter)	d = {'med':10, 'ali':20, 'lilia':5} sorted(d) # donne ['ali', 'lilia', 'med'] sorted(d, reverse = True) # donne l'ordre inverse['med', 'lilia', 'ali']
Parcours d'un itérable	for item in iter : #opérations manipulant item	lst = [1,2,5] for e in lst: print(e)
Énumération des éléments d'un itérable	enumerate (iter) enumerate (iter, start_index)	ch = "cem" for i, v in enumerate(ch): print(i, v) # donne en sortie 0 c 1 e 2 m

VII.2 Les séquences

Les itérables ordonnés, appelés encore séquences (tuple, liste et chaîne), conservent l'ordre dans lequel les éléments ont été créés. Ainsi, chaque élément d'une séquence possède un indice indiquant sa position relative par rapport aux autres éléments. Les indices commencent à partir de **0** pour un parcours de la gauche vers la droite.

Un intervalle défini par **range**, également considéré comme une séquence, est un itérable ou conteneur formé par une succession d'entiers.

range(debut, fin, pas) génère l'intervalle [debut, fin[par pas entier relatif égal à pas.

VII.2.1 Opérations communes aux séquences

Le tableau suivant liste quelques opérations communes à toutes les séquences.

Description	Syntaxe	Exemples / Remarques
Indexage simple	<code>s[i]</code> avec $0 \leq i \leq \text{len}(s) - 1$ ou $-\text{len}(s) \leq i \leq -1$	<code>s = 'python'</code> <code>s[0]</code> # donne 'p' <code>s[len(s)-1]</code> # ou <code>s[-1]</code> donne 'n' <code>s[len(s)]</code> # <code>IndexError</code>
Morcelage (Slicing)	# fin non inclus <code>s[dep:fin:pas]</code> # pas par défaut égal à 1 <code>s[dep:fin]</code> # du début avec pas=1 <code>s[:fin]</code> # jusqu'à la fin avec pas=1 <code>s[dep:]</code>	<code>s = 'python'</code> <code>s[::-1]</code> # donne 'nohtyp' <code>s[:2]</code> # donne 'py' <code>s[2:]</code> # donne 'thon' <code>s[:2]</code> # donne 'pto'
Concaténation	<code>s1 + s2</code>	<code>[1,2] + [3]</code> # donne <code>[1,2,3]</code> <code>(1,2) + (3,)</code> # donne <code>(1,2,3)</code> <code>'med' + 'ali'</code> # donne 'medali'
Répétition	<code>n * s</code> # ou bien <code>s * n</code> avec n entier Naturel	<code>[0] * 3</code> # donne <code>[0,0,0]</code> <code>(1,2) * 2</code> # donne <code>(1,2,1,2)</code> <code>'bon' * 2</code> # donne 'bonbon'
Comptage du nombre d'occurrences	<code>s.count(item)</code>	<code>lst = [0,1,0,1,1,1]</code> <code>lst.count(1)</code> # donne 4 <code>ch = "ali baba"</code> <code>ch.count("a")</code> # donne 3
Retour de l'index d'un item Déclenche <code>ValueError</code> si item not in s.	# première occurrence <code>s.index(item)</code> # à partir de start <code>s.index(item, start)</code>	<code>lst.index(0)</code> # donne 0 <code>lst.index(0,1)</code> # donne 2 <code>lst.index(5)</code> # <code>ValueError</code>
Parcours d'un itérable indexé	for ind in range(len(iter)) : #opérations manipulant iter[ind]	<code>lst = [1,2,5]</code> <code>for i in range(len(lst)):</code> <code>print(lst[i])</code>

VII.2.2 La classe str

Une chaîne est une séquence non modifiable et ordonnée de caractères. Le tableau suivant résume les principales opérations applicables aux chaînes.

Description	Syntaxe	Exemples / Remarques
Création d'une chaîne de caractères	#chaîne simple: ("..." , '...') #chaîne multilignes: ("..." ... "..." , ''' ... ''')	s = "bonsoir" # ou encore s = 'bonsoir' s1="""chaîne multi- lignes""" #équivalente à s1 = 'chaîne\nmulti-\nlignes'
Retour d'une chaîne en minuscule identique à Ch	Ch. lower ()	Ch='fG15Z' Ch.lower() # donne 'fg15z'
Retour d'une chaîne en majuscule identique à Ch	Ch. upper ()	Ch='fG15Z' Ch.upper () # 'FG15Z'
Retour d'une liste des sous-chaînes en utilisant un séparateur sep Par défaut, le séparateur est le caractère espace	Ch. split (sep)	Ch='Epsilon 2018/2019' Ch.split() # donne ['Epsilon', '2018/2019'] Ch.split('0') # donne ['Epsilon 2', '18/2', '19']
Retour d'une chaîne identique à Ch en supprimant les occurrences d'un séparateur <i>sep</i> du début et de la fin si elles existent. Par défaut sep est l'espace.	Ch. strip (sep)	Ch="*la chaîne*" Ch.strip('*') # donne 'la chaîne'
Retour d'une chaîne de caractères résultante de la concaténation par sep des chaînes de l'itérable TypeError est généré si l'itérable contient un objet qui n'est pas de type str.	Sep. join (itérable)	L=['abc', '12', 'azer'] '/' .join(L) # donne 'abc/12/azer' ' ' .join(L) # donne 'abc 12 azer'
Retour d'une chaîne de caractères en remplaçant chaque '{' par les paramètres passés à la méthode (objets de classes quelconques).	Ch. format (paramètres)	Ch = 'Epsilon {}/{ }' Ch.format(2019,2020) # donne 'Epsilon 2019/2020'
Retour de l'Unicode (entier) du caractère passé en paramètre	ord (caractère)	ord('A') # 65 ord('a') # 97
Retour du caractère associé à la valeur UNICODE passée paramètre.	chr (entier)	chr(65) # donne 'A' chr(10) # donne '\n'

VII.2.3 La classe list

Une liste est une séquence, ordonnée et modifiable, d'éléments éventuellement hétérogènes séparés par une virgule et délimitée par une paire de crochets.

Le tableau suivant résume les principales fonctions/méthodes applicables aux listes.

Description	Syntaxe	Exemples
Création d'une liste vide	<code>L=[]</code> #ou <code>L=list()</code>	<code>L=[1, 'python', True, 1+1j]</code>
Création d'une liste à partir d'un itérable	<code>L=list(itérable)</code>	<code>list('ABCA')</code> # donne ['A','B','C','A'] <code>list({3, 'A',4})</code> # donne [3,4,'A'] <code>list((2,2,(1,2)))</code> # donne [2,2,(1,2)] <code>list({2 :45, 'x':[1,2]})</code> # donne [2,'x'] <code>list(145)</code> # TypeError
Ajout d'un objet à la fin d'une liste	<code>L.append (objet)</code>	<code>L.append(2020) ; L</code> #donne [1, 'python', True, (1+1j), 2020] #remplissage d'une liste par n chaînes <code>lst = list()</code> <code>for i in range(n):</code> <code>x = input("donner l[{}]:".format(i))</code> <code>lst.append(x)</code>
Ajout d'un objet x dans L à l'indice i	<code>L.insert(i,x)</code>	<code>L.insert(-1 , 'concours') ; L</code> # donne [1, 'python', True, (1+1j), 'concours', 2020]
Suppression de la première occurrence d'un l'objet x dans L ValueError si x not in L.	<code>L.remove(x)</code>	<code>L.remove(1+1j) ; L</code> # donne [1, 'python', True, 'concours', 2020]
Ajout d'éléments de l'itérable t à la fin de L	<code>L.extend(t)</code>	<code>L.extend({-1,2}) ; L</code> # donne [1, 'python', True, 'concours', 2020, 2, -1]
Suppression et retour du dernier élément de L	<code>L.pop()</code>	<code>L.pop()</code> # donne -1 <code>L</code> # donne [1, 'python', True, 'concours', 2020, 2]
Suppression et retour de l'élément d'indice i de la liste L	<code>L.pop(i)</code>	<code>L.pop(2)</code> # donne True <code>L</code> # donne [1, 'python', 'concours', 2020, 2]

VII.2.4 La classe tuple

Un tuple est une séquence, ordonnée et non modifiable, d'éléments éventuellement hétérogènes séparés par une virgule. Le tableau suivant résume les principales fonctions/méthodes applicables aux tuples :

Description	Syntaxe	Exemples
Création d'un tuple vide	<code>t=()</code> <code>#t=tuple()</code>	
Création d'un tuple d'un seul objet (singleton)	<code>t=objet</code> ,	<code>T=255,</code> <code>T</code> <code># donne (255,)</code> <code>T1=[12,5],1</code> <code>T1</code> <code>#donne ([12,5], 1)</code>
Création d'un tuple à partir d'un itérable	<code>l=tuple(itérable)</code>	<code>tuple({3,'A',4})</code> <code># donne (3,'A',4)</code> <code>tuple((2,2,(1,2)))</code> <code># donne (2,2,(1,2))</code>

VII.3 Les itérables non ordonnés

VII.3.1 La classe set

Un ensemble est une collection d'éléments distincts non ordonnés et encadrés par des accolades. La classe `set` représente la notion d'ensemble en mathématique. Un ensemble doit impérativement comporter des éléments non modifiables. La création d'ensembles de listes, de dictionnaires ou d'ensembles déclenche une erreur. Le tableau suivant résume les principales fonctions/méthodes applicables aux ensembles.

Description	Syntaxe	Exemples/ Remarques
Création d'un ensemble vide	<code>s = set()</code> NB : <code>s={}</code> ne crée pas un ensemble	
Création d'ensemble à partir d'un itérable <code>iter</code> ou avec les délimiteurs	<code>s = set(iter)</code> <code>s={e0, ...,en}</code>	<code>s = set("010101")</code> <code>s</code> <code># donne {'0','1'}</code> <code>s = set(3.25)</code> <code># TypeError</code>
Ajout d'un objet	<code>s.add(obj)</code>	<code>s = set()</code> <code>s.add(5)</code> ; <code>s</code> <code># donne {5}</code> <code>s.add("eps");</code> <code>s</code> <code># donne {5, "eps"}</code> <code>s.add([1,2,3])</code> <code>#TypeError</code>
Suppression d'un objet	<code>s.discard(x)</code> <code>s.remove(x)</code> <code>s-= {obj}</code>	<code>s = {1,7,2}</code> ; <code>s.remove(1)</code> <code>s</code> <code># donne {2,7}</code> <code>s.remove(5)</code> <code># KeyError</code> <code>s -= {2}</code> ; <code>s</code> <code># donne {7}</code>
Union : $s1 \cup s2$	<code>s1 s2</code> <code>s1.union(s2)</code>	<code>s = {1,2} {2,3}</code> <code>s</code> <code># donne {1,2,3}</code>
Intersection : $s1 \cap s2$	<code>s1 & s2</code> <code>s1.intersection(s2)</code>	<code>s = {1,2} & {2,3}</code> <code>s</code> <code># donne {2}</code>
Différence : $s1 \setminus s2$	<code>s1 - s2</code> <code>s1.difference(s2)</code>	<code>s = {1,2} - {2,3}</code> <code>s</code> <code># donne {1}</code>
Différence Symétrique $s1 \Delta s2$	<code>s1 ^ s2</code> <code>s1.symmetric_difference(s2)</code>	<code>s = {1,2} ^ {2,3}</code> <code>s</code> <code># donne {1,3}</code>
Inclusion : $s1 \subset s2$	<code>s1 <= s2</code>	<code>{1,2} <= {1,2,5}</code> <code># donne True</code> <code>{1,2} <= {1,2}</code> <code># donne True</code>
Inclusion stricte : $s1 \subsetneq s2$	<code>s1 < s2</code> <code>s1.issuperset(s2)</code>	<code>{1,2} < {1,2,5}</code> <code># donne True</code> <code>{1,2} < {1,2}</code> <code># donne False</code>

VII.3.2 La classe dict

Un dictionnaire est une structure modifiable et non ordonnée d'associations (clé : valeur). Les clés sont impérativement des objets non modifiables. L'utilisation de listes, de dictionnaires ou d'ensembles comme clé déclenche une erreur. Les valeurs sont des objets de classes quelconques.

Description	Syntaxe	Exemples / Remarques
Création d'un dictionnaire vide	<code>d = dict()</code> <code>d = {}</code>	
Création d'un dictionnaire à partir d'un itérable de couples ou directement	<code>d = dict(iter)</code> <code>d = {cl:v1 , ..., cn:vn}</code>	<code>t = [(5,2), "ab", [1,2]]</code> <code>dict(t) #donne {1:2, 5:2, 'a':'b'}</code> <code>d = {'a': 3, 'b': 5, 4j: 33}</code>
Test d'appartenance d'une clé k à un dictionnaire d	<code>k in d</code> <code>k not in d</code>	<code>33 in d</code> # donne False <code>'a' in d</code> # donne True
Accès à une entrée du dictionnaire d	<code>d[k]</code> <code>d.get(k)</code> <code>d.get(k, vdef)</code>	<code>d = {'a': 3, 'b': 5, 4j: 33}</code> <code>d['a']</code> # donne 3 <code>d['c']</code> # donne KeyError <code>d.get('c')</code> # None <code>d.get('c', 0)</code> #0
Mise à jour / insertion d'une entrée dans le dictionnaire d	<code>d[k] = val</code> <code>d.update({k :val})</code>	<code>d = {}</code> <code>d['a'] = 33</code> # insertion <code>d</code> # donne {'a':33} <code>d['a'] = 22</code> # mise à jour <code>d</code> # donne {'a':22}
Copie de dictionnaire	<code>d.copy()</code>	<code>d = {1:2}</code> <code>d1 = d.copy()</code> <code>d[1] = {2} ; d1</code> # donne {1:{2}} <code>d</code> # donne {1:2}
Suppression d'une entrée de à partir de sa clé	<code>d.pop(k)</code> <code>del d[k]</code>	<code>d = {1:2, 'a':10}</code> <code>del d[1]; d</code> # donne {'a':10} <code>d.pop('a')</code> # donne la valeur 10 <code>d</code> # donne {}
Parcours des clés d'un dictionnaire	<code>for k in d:</code> traitements #ou encore <code>for k in d.keys()</code> traitements	<code>d = {1:2, 'a':10}</code> <code>for k in d:</code> print(k)
Parcours des valeurs d'un dictionnaire	<code>for v in d.values():</code> traitements	<code>for v in d.values():</code> print(v)
Parcours des couples clé, valeur d'un dictionnaire	<code>for k, v in d.items():</code> traitement	<code>for k, v in d.items():</code> print(k,v)

VIII Construction des itérables en compréhension

La construction des itérables en compréhension permet de générer des conteneurs de façon concise et élégante. On peut créer des listes en compréhension, des tuples en compréhension, des ensembles en compréhension et des dictionnaires en compréhension à l'aide du constructeur de la classe ou les délimiteurs des itérables mutables.

VIII.1 Listes en compréhension

```
[expression for indice_1 in iterable_1 if condition_1 ...
    for indice_n in iterable_n if condition_n ]
```

ou encore

```
list(expression for indice_1 in iterable_1 if condition_1 ...
    for indice_n in iterable_n if condition_n )
```

- Les conditions **if** sont facultatives (conditions de filtrage).
- L'ordre d'imbrication des boucles est de la gauche vers la droite.
- On peut utiliser le constructeur `list` à la place des délimiteurs.
- **expression** peut elle même être un itérable en compréhension.

Exemples

```
>>> L= [ i for i in range(1, 21, 2) ] ; L
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
>>> L=[ (i,j) for i in range(1,3) for j in range(1,4) ] ; L
[(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3)]
>>> L=[i*j for i in range(1,5) for j in range(1,5) ] ; L
[1, 2, 3, 4, 2, 4, 6, 8, 3, 6, 9, 12, 4, 8, 12, 16]
>>> L=[(i,j) for i in range(1,5) if i%2 for j in range(1,5) if (i+j)%2==0]; L
[(1, 1), (1, 3), (3, 1), (3, 3)]
>>> [[(100*i+j)//2 if j%2==0 else 100*i+j for j in range(1,i+1)] for i in
range(1,5)]
[[101], [201, 101], [301, 151, 303], [401, 201, 403, 202]]
```

VIII.2 Tuples en compréhension

```
tuple( expression for indice_1 in iterable_1 if condition_1 ...
    for indice_n in iterable_n if condition_n )
```

Attention : sans le constructeur `tuple`, la construction en compréhension produit un générateur et non pas un tuple.

Exemple

```
tuple((i,j) for i in range(1,5) if i%2 for j in range(1,5) if (i+j)%2==0 )
((1, 1), (1, 3), (3, 1), (3, 3))
```

VIII.3 Ensembles en compréhension

```
set(expression      for indice_1 in iterable_1 if condition_1  ...
                  for indice_n in iterable_n  if condition_n )
```

ou encore

```
{ expression      for indice_1 in iterable_1 if condition_1  ...
  for indice_n in iterable_n  if condition_n }
```

Exemple

```
>>> e={ (x,y) for x in range(5) if x%2 for y in range(5) if y%2==0 }
# ou bien e=set((x,y) for x in range(5) if x%2 for y in range(5) if y%2==0)
>>> e
{(1, 2), (3, 2), (3, 0), (1, 4), (1, 0), (3, 4)}
```

VIII.4 Dictionnaires en compréhension

```
{cle: valeur      for indice_1 in iterable_1 if condition_1  ...
                  for indice_n in iterable_n  if condition_n }
```

ou encore

```
dict((cle,valeur)  for indice_1 in iterable_1 if condition_1  ...
      for indice_n in iterable_n  if condition_n )
```

Exemple:

```
>>>d={ x:y for x in range(10) if x%2 for y in range(10) if y%2==0}
#ou bien d=dict((x,y) for x in range(10) if x%2 for y in range(10) if y%2==0)
>>>d
{1: 8, 3: 8, 9: 8, 5: 8, 7: 8}
```

IX Import de bibliothèques ou modules

Il y a trois manières d'import de modules :

– 1^{ère} manière d'import :

- Import du module désigné par *nom_module* mais pas les fonctions qui y sont contenues :

```
import nom_module
```

Exemples : import math # import du module math
import random # import du module random

- L'appel de chaque fonction du module se fait par :

```
nom_module.fonction(...)
```

Exemples : math.ceil(7.8989) # partie entière supérieure qui est 8
random.randint(0,10) # retourne un entier aléatoire entre 0 et 10

– 2^{ème} manière d'import :

- Utilisation d'un alias :

```
import nom_module as alias
```

Exemple : `import math as m`

- L'appel de chaque fonction du module se fait par :

```
alias.fonction(...)
```

Exemple : `m.sqrt(4)` # retourne la racine carrée de 4 : 2.0

– 3^{ème} manière :

- Accès direct à toutes les fonctions du module par :

```
from nom_module import *
```

Exemple : `from math import *`

- Accès direct à une ou quelques fonctions du module par :

```
from nom_module import nom_fonc1,...,nom_fonc_p
```

Exemple : `from math import pi,sqrt,ceil`

Dans les deux cas, l'appel de chaque fonction du module se fait directement par son nom : `nom_fonction(...)`

Exemple : `sqrt(5)` #donne 2.23606797749979

X Les sous-programmes

Une fonction est un bloc d'instructions indentées (corps de la fonction) destiné à réaliser une opération. Elle peut recevoir des paramètres et renvoie éventuellement un résultat. L'appel d'une fonction Python retourne toujours un objet, par défaut `None`. Dans le corps de la fonction ce retour est spécifié à l'aide de l'instruction **return**.

Le tableau suivant résume les concepts de base associés aux fonctions Python.

Description	Syntaxe	Exemples(s) / Remarques
Définition d'une fonction où les paramètres sont optionnels	def nom_f(paramètres) : corps de nom_f	#fonction avec paramètres def f(x,y) : import math return x**2 + math.sin(y) #fonction sans paramètres def g(): while True: s = input("donner une chaîne alphabétique : ") if s.isalpha(): return s
Appel d'une fonction : - Arguments positionnels - Arguments nommés (ordre arguments n'a plus d'importance) Les paramètres nommés doivent apparaître à la fin	nom_f(arguments) nom_f(n_arg1=v1 , ..., n_argp=vp)	#arguments positionnels print(f(3,2)) #arguments nommés r = f(y = 2, x = 3)
Retour d'une fonction L'instruction return permet de retourner un résultat à l'appelant d'une fonction. Cette instruction engendre la sortie immédiate du code de la fonction et le retour vers l'appelant.	return expression #Une fonction sans return retourne l'objet None	#Définition def f(x) : if x > 0 : return x**0.5 #Appel print(f(4)) # affiche 2.0 print(f(-4)) # affiche None
Paramètres optionnels Les paramètres optionnels sont associés à une valeur par défaut (évaluée une seule fois au moment de la définition de la fonction). Si l'argument associé au paramètre n'est pas indiqué lors de l'appel, Python utilise la valeur par défaut pour substituer le paramètre optionnel. Les paramètres optionnels doivent apparaître à la fin de la séquence des paramètres.	def f(x,y,z=val_def) : corps de f	# cette déclaration génère SyntaxError def h(x, y = 5, t) : corps # déclaration de f def f(x, y = 6) : return x + y #Appel de f print(f(1,3)) # affiche 4 print(f(3)) # affiche 9
Fonctions lambda Elle permet de créer une expression de type fonction.	lambda paramètres : résultat	f = lambda x : x**2 #équivalente à def f(x) : return x**2

XI Gestion des exceptions

Les exceptions sont des objets spéciaux permettant de représenter une erreur d'exécution dans un script Python. Il existe plusieurs types d'erreurs : `TypeError`, `ValueError`, `IndexError`. Ces derniers sont des classes Python héritant toutes de la classe de base `Exception` permettant de représenter une Erreur quelconque. La table suivante résume les opérations de gestion d'erreur.

Description	Syntaxe	Exemple(s) / Remarque(s)
Déclencher une exception	<code>raise Exception(" message ")</code>	<code>raise Exception()</code>
Prévoir une exception	<pre> if cond_erreur : raise Exception("message") ou bien assert not cond_erreur, "message" </pre>	
Intercepter une exception	<pre> try : bloc except : bloc_cas_erreur else : bloc_cas_pas_erreur finally : bloc_tous_les_cas </pre>	<pre> #saisie d'un entier positif (avec gestion d'erreur) def f(): while True: try: x=int(input('x')) except: print ('erreur') else: if x>0: return x </pre>

XII Gestion des fichiers

Un fichier est une collection d'informations, généralement structurées d'une manière ou d'une autre, placées sur une mémoire de masse.

XII.1 Fichiers textes

Un fichier texte contient une succession de caractères séquentiels. Lors de l'ouverture d'un fichier texte, un encodeur assure la conversion des octets lus/écrits en lettres UNICODE. Voici les principales fonctions/méthodes pour le traitement des fichiers.

Fonctions /Méthodes	Description/Exemples
<code>f=open(nom_du_fichier,'r')</code>	Ouvre le fichier <code>nom_de_fichier</code> (donné sous la forme d'une chaîne de caractères indiquant son emplacement) en lecture (<code>r</code> comme <code>read</code>). Le fichier doit exister et seule la lecture est autorisée. L'ouverture d'un fichier inexistant génère une erreur. L'utilisation de la clause <code>try ... except</code> est indiquée pour gérer cette situation.
<code>f=open(nom_du_fichier,'w')</code>	Ouvre le fichier <code>nom_de_fichier</code> en écriture (<code>w</code> comme <code>write</code>). Si le fichier n'existe pas, il est créé, sinon il est écrasé. <code>f</code> est le fichier logique associé au fichier physique donné par nom du fichier .
<code>f=open(nom_du_fichier,'a')</code>	Ouvre le fichier <code>nom_de_fichier</code> en ajout (<code>a</code> comme <code>append</code>). Identique au mode ' <code>w</code> ', sauf que si le fichier existe, il n'est pas écrasé et l'ajout se fait en fin du fichier.
<code>f.close()</code>	Ferme un fichier ouvert par la fonction <code>open</code> .
<code>f.read()</code>	Fournit l'intégralité du contenu du fichier sous forme de chaîne de caractères (à ne réserver qu'aux fichiers de taille raisonnable).
<code>f.readlines()</code>	Fournit la <u>totalité</u> du contenu du fichier <u>sous forme d'une liste de chaînes de caractères</u> . Le caractère de saut de ligne ' <code>\n</code> ' présent à la fin de chaque chaîne indique la fin de chaque ligne.
<code>f.readline()</code>	Fournit <u>la ligne courante</u> sous forme d'une chaîne de caractères et place le curseur sur la ligne suivante. Elle renvoie la chaîne vide si fin du fichier.
<code>f.write(s)</code>	Écrit la chaîne <code>s</code> à partir de la position courante du fichier.
<code>f.writelines(T)</code> T doit être une chaîne de sous chaînes ou un itérable de chaînes	Écrit l'ensemble des éléments de <code>T</code> dans le fichier <code>f</code> comme des lignes successives. Le caractère de saut de ligne ' <code>\n</code> ' présent à la fin de chaque chaîne indique la fin de chaque ligne.
Parcours d'un fichier	
Parcourir un fichier ouvert en lecture se fait comme le parcours d'un itérable : parcourir ses lignes avec une boucle <code>for</code> sans passer par <code>readlines</code> . <pre>for ligne in f : [instructions]</pre>	<pre>try: f = open("nomf.txt","r") except: print(" fichier introuvable !") else: for ligne in f: print(ligne) f.close()</pre>

On n'est jamais à l'abri d'une erreur quand on manipule des fichiers et qu'on oublie de les fermer. Le mot réservé **`with`** du langage python, permet d'éviter cette situation et dont voici sa syntaxe générale :

```
with open (mon_fichier, mode) as variable :  
    # Opérations de manipulation du fichier
```

L'exemple précédent s'écrit alors :

```
try:  
    with open ('fp.txt','r') as f :  
        for ligne in f:  
            print(ligne)  
except:  
    print(" fichier introuvable !")
```

XII.2 Fichiers binaires

Dans un fichier binaire les informations sont transférées de la RAM vers le support de stockage sans aucun formatage (aucun encodeur n'est utilisé). Pour les fichiers binaires, le mode d'ouverture est suffixé par 'b' : 'wb', 'rb' et 'ab'. La manipulation d'un fichier binaire n'est pas une fin en soi et cela ne présente aucun intérêt pour l'utilisateur. L'intérêt à ce niveau est de pouvoir utiliser le module **pickle** permettant de sérialiser/désérialiser des objets (entier, réel, chaîne, liste, dictionnaire, etc.) dans leur format et non pas sous forme de suite de caractères.

Sérialiser un objet, à l'aide de la méthode **dump**, consiste à le transformer en une suite d'octets afin de pouvoir le stocker dans un fichier binaire. Désérialiser, à l'aide de la méthode **load**, consiste à récupérer les objets dans leur format initial.

```
>>>import pickle
>>>pickle.dump(objet, fichier_binaire)
>>>objet = pickle.load(fichier_binaire)
```

Remarque : **load** génère une erreur si plus d'objets à lire, l'utilisation du `try ... except` s'impose.

XIII La récursivité

Une fonction récursive est une fonction dont le corps contient un ou plusieurs appels à elle-même.

Un traitement récursif nécessite de déterminer :

- le cas trivial ou encore le cas d'arrêt
- le traitement récurrent et de s'assurer que ce dernier atteindra le cas d'arrêt (terminaison).

Dans le tableau suivant, on donne deux exemples de problèmes résolus de façon récursive.

Problème	Démarche	Solution récursive
Calcul de la factorielle de n $n! = \prod_{i=1}^n i$	On raisonne par récurrence sur n ▪ Cas trivial : $0! = 1$ ▪ $n! = n \times (n-1)!$	<pre>def factorielle(n): if n<=1: return 1 #cas trivial else: return n*factorielle(n-1)</pre>
Tri Fusion	Le tri fusion est un tri récursif qui applique le principe "<i>diviser pour régner</i>". Pour une liste L, le principe du tri fusion est le suivant : ➤ si L est de taille 1, elle est triée et on la retourne (cas trivial) ; ➤ sinon on divise L en deux sous-listes L1 et L2 presque égales ($L1[0:len(L)//2]$ et $L2[len(L)//2:len(L)]$) : ⇒ on applique récursivement l'algorithme de tri par fusion à L1 puis à L2 ⇒ on fusionne les L1 et L2 triées. (La fusion est également effectuée de manière récursive où le cas trivial est l'épuisement de l'une des 2 listes et on retourne l'autre).	<pre>def fusion_Rec(L1,L2): if L1==[]: return L2 if L2==[]: return L1 if L1[0]<L2[0]: return [L1[0]]+fusion_Rec(L1[1:],L2) else: return [L2[0]]+fusion_Rec(L1,L2[1:]) def tri_fusion_Rec(L): if len(L)<=1: return L L1=[L[x] for x in range (len(L)//2)] L2=[L[x] for x in range (len(L)//2, len(L))] Return fusion_Rec(tri_fusion_Rec(L1, tri_fusion_Rec(L2))</pre>

XIV Complexité algorithmique

XIV.1 Calcul du coût

Le temps d'exécution d'un programme dépend fortement de la machine qui l'exécute, du langage dans lequel il est écrit et de l'algorithme sous-jacent. Pour des données de grandes tailles, l'algorithme est la cause déterminante du temps d'exécution d'un programme. Des méthodes rationnelles permettent de calculer le coût d'un programme exprimé en nombre d'opérations élémentaires exécutées par la machine afin d'obtenir le résultat final. Généralement, ce nombre est fonction de la taille n des données à traiter.

Le coût se calcule comme suit :

- Coût d'une opération élémentaire **op** = 1.
- Coût d'une séquence de **k** instructions **p_1 ; p_2 ; ... p_k** = $\sum_{i=1}^k \text{coût}(p_i)$.
- Coût d'une structure conditionnelle simple **if c: p else: q** $\leq \text{coût}(c) + \max(\text{coût}(q), \text{coût}(p))$.
- Coût d'une structure conditionnelle complète **if c_1: p_1 elif c_2: p_2 ... elif c_i: p_i ... else: p_{k+1}**
 $= (\sum_{i=1}^k \text{coût}(c_i)) + \max_{1 \leq i \leq k+1} \text{coût}(p_i)$.
- Coût d'une boucle **for i in iterable: p_i** = $\sum_{i \in \text{itérable}} \text{coût}(p_i)$
- Coût d'une boucle **while c: p** est le plus difficile à exprimer (nombre d'itérations non connu à priori)
 $\rightarrow$ majorer le nombre de répétitions $\Rightarrow \text{NombreRépétitions} * (\text{coût}(c) + \text{coût}(p)) + 1 * \text{coût}(c)$

Exemple : Calcul x^n

- *Approche naïve* :

```
def puiss1(x,n):
    r = 1 #1 affectation
    for i in range(n): # n affectations
        r*= x          # 1 affectation + 1 multiplication
    return r          # 1 retour
```

Le Coût total = $f(n) = 2n + 2$.

- *Exponentiation rapide* :

```
def puiss2(x,n):
    r= 1      # 1 affectation
    u= x      # 1 affectation
    while n != 0: # (int(log2(n)) + 1) comparaisons
        if n % 2 == 1: #2 = modulo + comparaison
            r*= u      #2 = multiplication + affectation
            u*= u      #2 = multiplication + affectation
        n//= 2         #2 = division + affectation
    return r          #1 retour
```

Le Coût total = $f(n) = 9 E[\log_2(n)] + 3$.

XIV.2 Complexité asymptotique

La complexité asymptotique consiste à estimer l'ordre de grandeur du nombre d'opérations effectuées par un programme sans effectuer un décompte minutieux. Cette complexité est exprimée en Grand-O (notation de Landau).

Soit $f, g : \mathbb{N} \rightarrow \mathbb{R}_+^*$ deux fonctions. $f(n)$ est dite $O(g(n)) \Leftrightarrow f$ est dominée par g quand $n \rightarrow +\infty$.

- $f(n)$ est $O(g(n))$ et c une constante, alors :
 - $\Rightarrow c + f(n)$ est aussi $O(g(n))$.
 - $\Rightarrow c f(n)$ est aussi $O(g(n))$.
 - $\Rightarrow$ Les facteurs constants sont ignorés dans le calcul de la complexité asymptotique.
- $f_1(n)$ est $O(g_1(n))$ et $f_2(n)$ est $O(g_2(n))$, alors :
 - $\Rightarrow f_1(n) + f_2(n)$ est $O(g_1(n) + g_2(n))$.
 - $\Rightarrow f_1(n)f_2(n)$ est $O(g_1(n)g_2(n))$.
- $f_1(n)$ est $O(f_2(n))$ et $f_2(n)$ est $O(f_3(n))$, alors :
 - $\Rightarrow f_1(n)$ est $O(f_3(n))$.

XIV.3 Classes de complexité asymptotique

Nom de la classe	Complexité	Exemples
Constante	$O(1)$	▪ Accès à un élément d'une liste $l[i]$. ▪ Additionner multiplier un nombre constant d'objets.
Logarithmique	$O(\log(n))$	▪ Recherche dichotomique $n = \text{len}(lst)$. ▪ Méthode de bissectrices pour trouver le zéro d'une fonction $n = \frac{b-a}{\epsilon}$. ▪ Exponentiation rapide. ▪ Trouver l'écriture d'un entier n dans une base b.
Linéaire	$O(n)$	▪ Trouver le maximum/minimum d'une liste $n = \text{len}(lst)$. ▪ Tester l'appartenance d'un élément dans une liste $n = \text{len}(lst)$. ▪ Produit scalaire de deux vecteurs.
Quasilinéaire	$O(n \log(n))$	▪ Trier une liste $lst.sort() : n = \text{len}(lst)$. ▪ Tri fusion / Tri rapide
Quadratique	$O(n^2)$	▪ Tri par sélection/insertion/bulles $n = \text{len}(lst)$. ▪ Remplissage d'une matrice carrée. ▪ Somme de deux matrices carrées
Polynomiale	$O(n^p)$ où $p \geq 2$	▪ Multiplication de deux matrices $O(n^3)$. ▪ Réduction de Gauss. ▪ Inversion d'une matrice.
Exponentielle	$O(c^n)$ où $c > 1$	▪ Trouver tous les sous-ensembles d'un ensemble de taille n. ▪ Trouver un mot de passe de taille n par brute force. ▪ Afficher toutes les valeurs binaires qui s'écrivent sur n bits.
Factorielle	$O(n!)$	▪ Résoudre le problème du voyageur du commerce (avec $n =$ nombre de villes) par brute force. ▪ Trier une liste de taille n par brute force.

Exemples :

- $4n + 100000000$ est $O(n)$.
- $100n^2 + n^{10} + 300000n$ est $O(n^{10})$.
- $2^n + n^{10000000}$ est $O(2^n)$.
- $2^n + n!$ est $O(n!)$.

Bibliographie et Netographie

Ouvrages

Learning Python, Mark Lutz, Edition O'Reilly, Juin 2013, ISBN: 9781449355722.

Informatique pour tous en classes préparatoires aux grandes écoles, Benjamin Wack & al., Edition Eyrolles, Août 2013, ISBN: 978-2-212-13700-2.

Supports de Cours

Support de Cours de H.Filali et H.Bouzidi, Rappel, pour 2^{ème} année préparatoire sections MP-PC-T, IPEIM, Année universitaire 2019-2020.

Support de Cours de Laila Hammami pour 1^{ère} année préparatoire sections MP-PC-T, IPEIN, Année universitaire 2019-2020.

Sites Web :

<https://www.python-course.eu/>

<https://www.ics.uci.edu/~brgallar/index.html>