

Résumé du cours : Notions de base

HOUCMEDDINE FILALI

HOUSSEMEDDINE BOUZIDI

Opérations de base

<i>Opérations d'entrée sortie</i>		
Description	Syntaxe	Exemple(s) & Remarques
Sortie	<code>print(expr1, ... , exprn)</code>	<code>print("Welcome to Python")</code>
Entrée	<code>nom_var = input()</code> <code>nom_var = input(" message ")</code>	<code>nom = input('donner votre nom : ')</code> <code>n = int(input(' n = '))</code> <code>x = float(input(' x = '))</code>
Affectation	<code>nom = expression</code>	<code>x = 5 #simple</code> <code>x = y= z = t = 5 #simultanée</code> <code>x, y, z = 1, 2, 5 #multiple</code>

<i>Opérations arithmétiques</i>		
Description	Syntaxe	Exemple(s) & Remarques
Parenthèses	<code>()</code>	<code>x+2**3 # $x + 2^3$</code> <code>(x+2)**3 # $(x + 2)^3$</code>
Puissance	<code>**</code>	<code>1j**2 ##-1+0j</code>
Signes	<code>+ -</code>	
Opérateurs multiplicatifs	<code>*</code> <code>/ # division flottante</code> <code>// # division entière</code> <code>% # modulo</code>	<code>4/2 #2.0 (float)</code> <code>4//2 #2 (int)</code> <code>4%2 #0</code>
Opérateurs additifs	<code>+ -</code>	

Opérations relationnelles

Description	Syntaxe	Exemple(s) & Remarques
Comparaison d'identité	<code>is</code> <code>is not</code>	<pre> 1 = [1,2,3] l1 = 1 l2 = l1.copy() 1 is l1 #True id(l) == id(l1) 1 is l2 #False id(l) != id(l2) </pre>
Comparaison de contenu	<code>==</code> <code>!=</code>	<pre>1 == l2 #True</pre>
Comparaison riche	<code><</code> <code>></code> <code><=</code> <code>>=</code>	<pre>(1, 2, 3) < (5, -10, 3) # True</pre>

Opérations logiques

Description	Syntaxe	Exemple(s) & Remarques
Produisent un booléen (instance de la classe <code>bool</code>)	<code>and</code> <code>or</code> <code>not</code>	



Une classe (type) décrit les attributs (les données) et les opérations (méthodes) de ses instances.

Description	Syntaxe	Exemple(s) & Remarques
Classe de l'objet spécial None : cette classe possède une seule instance elle est utilisée par Python pour indiquer l'absence de valeur de retour pour certaines fonctions.	<code>NoneType</code>	<pre> type(None) #NoneType x = print("Hello") x is None # True </pre>
Booléens bool : cette classe possède deux instances True & False qui représentent les valeurs de vérités.	<code>bool</code>	<pre> type(True) ##bool </pre>
Entiers relatifs ($\mathbb{Z}$)	<code>int</code>	<pre> x = int('52') x = 53 # littéral décimal x = 0b1101 # littéral binaire x = 0xabf5 # littéral hexadécimal x = 0o733 # littéral octal int(22.5) #22 </pre>
Nombres flottants : sous ensemble de $e(\mathbb{R})$ codées selon le standard IEEE754	<code>float</code>	<pre> x = float('22.2') x = 0.0001 # notation fractionnaire x = 1e-3 # notation exponentielle </pre>
Nombres complexes : sous ensemble de $(\mathbb{C})$: chaque instance est composée de deux attributs de type <code>float</code> : real qui décrit la partie réelle et imag qui décrit la partie imaginaire.	<code>complex</code>	<pre> z = 1+2j z2 = complex(3, 2) z1.real #1.0 (float) z2.imag #2.0 (float) z2.conjugate() #3-2j (complex) z = 1+j #NameError j (identificateur) z = 1+ 1j # correct : 1j unité imaginaire. </pre>

Un bloc de code est un groupe d'instructions Python précédée par le caractère « : » où chaque instruction du bloc est **indentée**. Les structures de contrôle permettent de changer le flot d'exécution d'un programme.

Description	Syntaxe	Exemple(s) & Remarques
Structure conditionnelle : permet de conditionner un ou plusieurs blocs d'instructions	<pre># obligatoire if cond : bloc1 # (optionnelle) autant de fois que nécessaire elif cond2 : bloc2 ... # (optionnelle) une seule fois à la fin else : blocd</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 Tantque : permet d'exécuter un bloc tant que la condition de la boucle est vraie, la condition est toujours testée avant l'exécution du bloc de la boucle.	<pre>while cond : bloc</pre>	$\forall x, y \in \mathbb{Z}^* \text{ pgcd}(x, y) = \begin{cases} y & \text{si } y x \\ \text{pgcd}(y, x \bmod y) & \text{sinon.} \end{cases}$ <pre>while x % y != 0 : x, y = y, x % y</pre> $(u_n)_{n \in \mathbb{N}} = \begin{cases} u_1 = 1 \\ u_n = u_{n-1}^2 + u_{n-1} \end{cases}$ trouver le premier terme u_k tel que $u_k > 10$ <pre>u = 1 while u <= 10 : u = u**2+u</pre>
Boucle conditionnelle Répéter ... jusqu'à cond : permet d'exécuter un bloc jusqu'à ce que la condition d'arrêt devienne False . La condition d'arrêt est toujours testée après l'exécution du bloc de la boucle.	<pre>while True : bloc if cond : break</pre>	<pre>#saisie d'un réel entre 0 et 1 while True : p = float(input(' p= ')) if 0 <= p <= 1: break</pre>

Boucle **Pour chaque élément de** : Permet d'exécuter le bloc pour chaque élément de l'itérable source. La classe `range` permet de générer un itérable formé par une succession d'entiers `int`.

```
range(n) # {0...n-1}
range(start,end) # {start...end-1}
range(start, end, step) # {start, start+step,...v}
# tel que  $|v - end| \leq step$ 
```

```
for v in itérable:
    bloc
```

```
# calcul de la factorielle de n
f = 1
for v in range(1,n+1):
    f *= v
```



La classe `str` : Une chaîne est une séquence non modifiable et ordonnée de caractères. Le tableau suivant résume les principales opérations applicables sur les chaînes.

Description	Syntaxe	Exemple(s) & Remarques
Création d'une chaîne de caractères	<pre>#chaîne simple ch = "Welcome" ch = 'Welcome' #idem #chaîne multilignes ch = """ une chaîne qui s'étale sur plusieurs lignes """</pre>	
Retourne une copie de ch en lettres minuscules	<code>ch.lower()</code>	<pre>ch='fG15Z' ch.lower() # 'fg15z'</pre>
Retourne une copie de ch à ch en lettres majuscule	<code>ch.upper()</code>	<pre>ch="fG15Z" ch.upper() # "FG15Z"</pre>
Retourne une liste des sous-chaînes obtenues en découpant ch autour du motif séparateur sep . Par défaut, les lettres transparentes (espace, retour à la ligne, retour charriot, tabulation etc.) sont considérées comme séparateur.	<code>ch.split(sep)</code>	<pre>ch='Epsilon 2018/2019' ch.split() #['Epsilon', '2018/2019'] ch.split('0') #['Epsilon 2', '18/2', '19']</pre>
Retourne une chaîne de caractères résultante de la concaténation des chaînes de l'itérable src intercalées par le motif sep . (déclenche TypeError si l'itérable contient un objet non instance de la classe <code>str</code>).	<code>sep.join(src)</code>	<pre>l=['abc', '12', 'azer'] '/' .join(l) # 'abc/12/azer' '@' .join(l) # 'abc@12@azer'</pre>
Retourne une chaîne de caractères en remplaçant chaque paire d'accolades "{}" par l'argument correspondant (dans le même ordre) de la méthode (objets de classe quelconque).	<code>ch.format(paramètres)</code>	<pre>ch = 'Epsilon {}/{}' ch.format(2017,2018) # 'Epsilon 2017/2018' ch='Hou{}emeddine {}' ch.format('c', 'FILALI') # 'Houcemeddine FILALI' ch.format('ss', 'BOUZIDI') # 'Houssemmeddine BOUZIDI'</pre>
Retourne le code (int) du caractère passé en paramètre selon le standard UNICODE	<code>ord(caractère)</code>	<pre>ord('A') # 65 ord('Z') # 90 ord('a') # 97</pre>
Retourne le caractère qui correspond au code (int) UNICODE passée paramètre	<code>chr(entier)</code>	<pre>chr(65) # 'A' chr(65+25) # 'Z' chr(10) # '\n'</pre>

La classe `list` : Une liste est une séquence d'éléments séparés par une virgule et délimitée par une paire de crochets. (Ordonnée, Mutable (modifiable), Indexée et Hétérogène)

Description	Syntaxe	Exemple(s) & Remarques
Création d'une liste	<pre>#liste vide l=[] l=list() #liste d'éléments l = [obj1, ... , objk] l = list(src) # ou src une instance d'une classe itérable</pre>	<pre>list('ABCA') # ['A','B','C','A'] list({3,'A',4}) # [3,4,'A'] list((2,2,(1,2))) # [2,2,(1,2)] list({2:45,'x':[1,2]}) # [2,'x'] list(145) # TypeError</pre>
Ajouter un objet à la fin de la liste <code>l</code> Ajouter l'objet <code>x</code> dans la liste <code>l</code> à l'indice <code>i</code>. Ajouter les éléments de l'itérable <code>t</code> à la fin de la liste <code>l</code> Retourner et supprimer le dernier élément de la liste <code>l</code>. Retourner et supprimer l'élément d'indice <code>i</code> de la liste <code>l</code>. Supprimer le premier objet <code>x</code> trouvé dans la liste <code>l</code>. <code>ValueError</code> si <code>x not in l</code>.	<pre>l.append(objet) l.insert(i ,x) l.extend(t) l.pop() l.pop(i) l.remove(x)</pre>	
Retourner le nombre d'occurrence de l'objet <code>x</code> dans la liste <code>l</code>	<pre>l.count(x)</pre>	<pre>l = [11,[15,99],11] l.count(11) # 2 l.count(99) # 0 l.count([15,99]) # 1</pre>
Retourner l'indice de la première occurrence de <code>x</code> dans <code>l</code> . <code>ValueError</code> si <code>x not in l</code> .	<pre>l.index(x)</pre>	
Parcourir un itérable.	<pre># parcourir à travers les indices for i in range (len(l)) : traitement sur l[i] # parcourir les éléments d'une liste for e in l: traitement sur e</pre>	<pre># parcourir les indices et les éléments for i, e in enumerate(l): traitement</pre>

La classe **tuple** : Un tuple est une séquence d'éléments séparés par une virgule (Ordonné, Immuable (non modifiable), Indexé et hétérogène). Le tableau suivant résume les principales opérations applicables sur les tuples.

Description	Syntaxe	Exemple(s) & Remarques
Création d'un tuple	<pre>#tuple vide t=() t=tuple() #singleton t = obj, # tuple d'éléments l = (obj1, ... , objk) l = tuple(src) # ou src une instance d'une classe itérable</pre>	<pre>t=255, #(255,) t=[12,5], #([12,5],) tuple('ABCA')#('A','B','C','A') tuple({3,'A',4})#(3,'A',4) tuple((2,2,(1,2)))#(2,2,(1,2)) tuple({2 :45,'x' :[1,2]})#(2,'x') tuple(145)# TypeError</pre>
Retourner le nombre d'occurrences de l'objet <i>x</i> dans le tuple <i>t</i>	<code>t.count(x)</code>	
Retourner l'indice de la première occurrence de l'objet <i>x</i> dans <i>t</i> . ValueError si <code>x not in t</code> .	<code>t.index(x)</code>	
Test d'appartenance d'un élément <i>x</i> à un itérable <i>t</i>	<pre>x in t x not in t</pre>	<pre>5 in (1,5,[1,3]) #donne True 3 not in (1,5,[1,3]) #donne True</pre>

La classe `set` : Représente la notion mathématique d'ensemble (valeurs uniques non ordonnées). Doit impérativement comporter des éléments non modifiables (créer un ensemble comportant des listes, des dictionnaires ou des ensembles déclenche l'exception `TypeError`).

Description	Syntaxe	Exemple(s) & Remarques
Création d'un ensemble	<pre>s = set() s = {} # créé une nouvelle instance de la classe dict s = {obj1, ..., objk} s = set(src) #où src est un objet itérable s = {v for v in it}</pre>	<pre>txt = '010101' s = set(txt) # {'0', '1'} s = set(3.25) # TypeError</pre>
Attention : Il est interdit d'indexer un ensemble <code>s[i]</code> déclenche l'exception <code>TypeError</code>		
Copie d'un ensemble	<code>s.copy()</code>	<pre>s = {1,2} ; s1 = s.copy() ; s -= {2} s1 #{1,2} s#{1}</pre>
Ajouter un objet	<code>s.add(obj)</code>	<pre>s = set() s.add(5) #s = {5} s.add("eps") #s = {5, "eps"} s.add([1,2,3]) #TypeError</pre>
Supprimer un objet	<pre>s -= {obj} s.discard(x) s.remove(x) # déclenche KeyError si x not in s</pre>	<pre>s = {1,2} s.remove(5) #KeyError s.remove(1) s #{2}</pre>
$s1 \cup s2$	<code>s1 s2</code>	<pre>s = {1,2} {2,3} s #{1,2,3}</pre>
$s1 \cap s2$	<code>s1 & s2</code>	<pre>s = {1,2} & {2,3} s #{2}</pre>
$s1 \setminus s2$	<code>s1 - s2</code>	<pre>s = {1,2} - {2,3} s #{1}</pre>
$s1 \Delta s2$	<code>s1 ^ s2</code>	<pre>s = {1,2} ^ {2,3} s #{1,3}</pre>
$s1 \subset s2$	<code>s1 < s2</code>	<pre>{1,2} < {1,2,5} #True {1,2} < {1,2} #False</pre>
$s1 \subseteq s2$	<code>s1 <= s2</code>	<pre>{1,2} <= {1,2,5} #True {1,2} <= {1,2} #True</pre>

La classe `dict` : Permet de construire une table associative (clé $\mapsto$ valeur). Les clés sont impérativement des objet **non modifiables** (utiliser des listes, des dictionnaires ou des sets comme clé déclenche `TypeError`). Les valeurs sont des objets de classe quelconque.

Description	Syntaxe	Exemple(s) & Remarques
Création d'un dictionnaire	<pre>d = dict() d = {} #dict vide d = {clé1 : obj1, ... , clék, objk} d= dict(src) # où src itérable d= {k:v for k,v in it} # par compréhension</pre>	<pre>t = [(5,2), "ab", [1,2]] dict(t) #{1: 2, 5: 2, 'a': 'b'} clés = [1,2,3] vals = 'abc' dict(zip(clés, vals)) # {1: 'a', 2: 'b', 3: 'c'}</pre>
Accès à une entrée du dictionnaire	<pre>d[k] #KeyError si k not in d d.get(k) #None si k not in d d.get(k, vdef) # vdef si k not in d</pre>	<pre>d = {'a':3, 'b':5, 4j:33} d['a'] #3 d['c'] #KeyError d.get('c') #None d.get('c', 0) #0 d.get('a', 0) #3</pre>
Mettre à jour / insérer une entrée dans le dictionnaire	<pre>d[k] = val #mise à jour si k in d #si k not in d insertion d.update({k :val}) #idem</pre>	<pre>d = {} d["a"] = 33 #insertion d #{"a" :33} d["a"] = 22 #mise à jour d# {"a" :22}</pre>
Supprimer une entrée à partir de sa clé	<pre>d.pop(k) #KeyError si k not in d #renvoie la valeur associée à la clé k del d[k] # instruction ne renvoie rien</pre>	<pre>d = {1:2, "a":10} del d[1] d# {"a":10} d.pop("a") # 10 d# {}</pre>
Parcourir les clés d'un dictionnaire	<pre>for k in d: traitements</pre> <pre>for k in d.keys(): traitements</pre>	<pre>d = {1:2, "a":10} for k in d: print(k)</pre>
Parcourir les valeurs d'un dictionnaire	<pre>for v in d.values(): traitements</pre>	
Parcourir les couples clé, valeur d'un dictionnaire	<pre>for k, v in d.items(): traitement</pre>	<pre>for k, v in d.items(): print(k,v)</pre>

Description	Syntaxe	Exemple(s) & Remarques
Taille d'un itérable	<code>len(it)</code>	
Test d'appartenance d'un élément <code>x</code> à un itérable <code>src</code>	<code>x in src</code> <code>x not in src</code>	<code>5 in {0,1,2}</code> #donne <i>False</i> <code>"yth" in "Python"</code> # <i>True</i> <code>"yth" in ["Python", 5, 2]</code> # <i>False</i>
Parcours d'un itérable	<code>for v in src:</code> traitement sur <code>v</code>	
Valeur maximale d'un itérable, <i>TypeError</i> pour un itérable comportant des objets non comparables.	<code>max(it)</code> <code>max(it, key = fct_critère)</code> <code>max(v1, v2, ..., vn)</code>	<code>1 = (1+3j, 2+5j, 11j)</code> <code>max(1)</code> # <i>TypeError</i> <code>max(1, key = abs)</code> # <i>11j</i>
Valeur minimale d'un itérable. <i>TypeError</i> pour un itérable comportant des objets non comparables.	<code>min(it)</code> <code>min(it, key = fct_critère)</code> <code>min(v1, v2, ..., vn)</code>	<code>min(1, key= lambda z : z.imag)</code> # <i>1+3j</i>
Trier un itérable d'objets comparables (<i>TypeError</i> si l'itérable source comporte des objets non comparables). Renvoie une <i>list</i> contenant les éléments dans l'ordre.	<code>sorted(it)</code> <code>sorted(it, key = fct_critère)</code> <code>sorted(it, reverse = True)</code> <code>sorted(it, reverse = True, key = fct_critère)</code>	<code>d = {'med':10, 'ali':20, 'lilia':5}</code> <code>sorted(d)</code> # <i>['ali', 'lilia', 'med']</i> <code>sorted(d, reverse = True)</code> # <i>['med', 'lilia', 'ali']</i> <code>sorted(d, key = lambda k : d[k])</code> # <i>['lilia', 'med', 'ali']</i>
Zipper plusieurs itérables (premier avec premier, deuxième avec deuxième etc.)	<code>zip(it1, it2)</code> <code>zip(it1, ..., itn)</code>	<code>ch = "abc"; l = [1,2,3]</code> <code>list(zip(ch, l))</code> # <i>[('a',1), ('b',2), ('c',3)]</i>
énumérer les éléments d'un itérable. Renvoie un itérable de tuples (couples (indice, valeur) ou valeur est prise de l'itérable source.)	<code>enumerate(iterable, start = 0)</code>	<code>>>> l = ["med", "amilie", "may", "cem"]</code> <code>>>> list(enumerate(l))</code> <i>[(0, 'med'), (1, 'amilie'), (2, 'may'), (3, 'cem')]</i> <code>>>> for i, v in enumerate("bc", start = 1):</code> <code>print(i, v)</code> <i>1 b</i> <i>2 c</i>
Renvoie <i>True</i> lorsque toutes les valeurs de l'itérable source sont évaluées à <i>True</i> , <i>False</i> sinon.	<code>all(itérable)</code>	<code>>>> lst = [2, 1, 3, 0]</code> <code>>>> all(x > 0 for x in lst)</code> # $\forall x \in lst : x > 0 \rightarrow$ <i>False</i> <code>>>> all(x >= 0 for x in lst)</code> # $\forall x \in lst : x \geq 0 \rightarrow$ <i>True</i>
Renvoie <i>True</i> s'il existe un élément de l'itérable source qui s'évalue à <i>True</i> , renvoie <i>False</i> sinon.	<code>any(itérable)</code>	<code>>>> any(x < 0 for x in lst)</code> # $\exists x \in lst : x < 0 \rightarrow$ <i>False</i> <code>>>> any(x <= 0 for x in lst)</code> # $\exists x \in lst : x \leq 0 \rightarrow$ <i>True</i>
Renvoie la somme de tous les éléments de l'itérable source (doit contenir des objets numériques uniquement).	<code>sum(itérable)</code>	<code>>>> sum(lst)</code> # $\sum_{x \in lst} x \rightarrow 6$ <code>>>> sum(x**2 for x in range(10))</code> # $\sum_{i=0}^9 x^2 \rightarrow 285$

Description	Syntaxe	Exemple(s) & Remarques
Concaténation (uniquement pour les séquences (<i>str</i> , <i>tuple</i> , <i>list</i>))	<code>seq1 + seq2</code>	<code>(1,2) + (3,) # (1,2,3)</code> <code>"med" + " " + "ali" #'med ali'</code>
Répétition (uniquement pour les séquences)	<code>n * seq</code> <i># ou bien</i> <code>seq * n</code>	<code>[0] * 3 # [0,0,0]</code> <code>-6 * [1,2] # []</code> <code>"ab" * 3 #"ababab"</code>
Indexage simple (uniquement pour les séquences). Pour les <i>dict</i> indexage par clé.	<code>s[i]</code> avec $0 \leq i \leq \text{len}(s)-1$ pour un indexage à partir de la gauche ou bien $-\text{len}(s) \leq i \leq -1$ pour un indexage à partir de la droite <code>s[i] = obj</code> Valide uniquement pour les séquences <i>mutables</i>	<code>s = "python" ; s[-1] #"n"</code> <code>s[0], s[len(s)-1] #"p", "n"</code> <code>s[len(s)] #IndexError</code>
Morceler une séquence (slicing)	<code>s[dep:fin]</code> <code>s[dep:fin:pas]</code> <code>s[dep :]; s[:fin]; s[::pas]</code>	<code>s = "python"</code> <code>s[:-1] # "nohtyp"</code> <code>s[:2] #"py"</code> <code>s[2:] #"thon"</code>
Retourner le nombre d'occurrence de l'objet <i>x</i> dans la séquence <i>s</i>	<code>s.count(x)</code>	<code>l = [11,[15,99],11]</code> <code>l.count(11) # 2</code> <code>l.count(99) # 0</code> <code>l.count([15,99]) # 1</code>
Retourner l'indice de la première occurrence de <i>x</i> dans <i>s</i> . <i>ValueError</i> si <i>x</i> <i>not in s</i> .	<code>s.index(x)</code>	<code>>>> s = 'ali baba'</code> <code>>>> s.index('ba')</code> <code>4</code> <code>>>> s.index('ma')</code> <code>ValueError</code> <code>>>> t = (s, 3, (5,3))</code> <code>>>> t.index(3)</code> <code>1</code> <code>>>> t.index('ali baba')</code> <code>0</code> <code>>>> t.index('baba')</code> <code>ValueError</code>

Une fonction est un **bloc** d'instructions (corps de la fonction) destiné à réaliser une opération. Elle peut recevoir des arguments et renvoie un objet. 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 expression**. Le tableau suivant résume les concepts de base reliés aux fonctions Python.

Description	Syntaxe	Exemple(s) & Remarques
Définition d'une fonction	<pre>def nom_f(paramètres): corps de la fonction</pre>	<pre>def f(x,y) : import math z = x**2 + math.sin(y) return z</pre>
Appel d'une fonction	<pre>nom_f(arguments)</pre>	<pre>print(f(3,2)) #arguments positionnels r = f(y = 2, x = 3)#arguments par mot-clé #définition def f(x) : if x > 0 : return x**0.5 #appel print(f(4)) #2.0 print(f(-4)) #None #fonction sans paramètres def f() : for i in range(10**10) : return i print(f()) #0</pre>
Retour d'une fonction L'instruction return permet de transmettre un résultat à l'appelant de la fonction en cours. Cette instruction engendre la sortie immédiate du code de la fonction et le retour vers le code appelant.	<pre>return return expression</pre> la sortie d'une fonction sans return (ou bien avec un return mais sans expression de retour) engendre le retour de l'objet 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.	<pre>def f(x, y, z = val_def) : corps de la fonction def f(x, y = 5, t) : corps #SyntaxError</pre>	<pre>def f(x, y = 6) : return x + y print(f(1,3)) #4 print(f(3)) #9 print(f(y = 2, x = 1))#3</pre>
Fonctions lambda : permet de créer une expression de type fonction. Utilisée généralement pour les fonctions d'ordre supérieur : fonctions qui prennent des paramètres de type fonction exemple (sorted , list.sort , min , max etc.)	<pre>lambda paramètres : résultat</pre>	<pre>f = lambda x : x**2 #équivalente à def f(x) : return x**2 d = {'ahmed':20, 'lilia':0} sorted(d) #['ahmed', 'lilia'] sorted(d, key= lambda k : d[k]) #['lilia', 'ahmed'] l = [3+4j, 44+33j] min(l) #TypeError min(l, key= lambda z : z.real)#3+4j</pre>

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

Description	Syntaxe	Exemple(s) & Remarques
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).	<pre>def fct_recursive(params): if not cas_arret: #cas récursif traitement_récursif fct_recursive(...) else: traitement_cas_arrêt</pre>	Calcul de la factorielle de n. Cas trivial $0! = 1$ Cas récursif $n! = n \times (n - 1)!$ <pre>def fact(n): if n<=1: return 1 #cas trivial else: return n*fact(n-1)</pre> Recherche dichotomique d'un élément x dans une liste lst triée en ordre croissant. Cas trivial : si liste vide alors résultat = False, sinon si l'élément du milieu m contient x alors résultat = True. Cas récursif : si $lst[m] > x$ alors effectuer la recherche dans la moitié gauche de la liste $0 \dots m - 1$. Si $lst[m] < x$ alors effectuer la recherche dans la moitié droite de la liste $m + 1 \dots len(lst)$. <pre>def bin_search(x,l): if len(l) == 0: return False else: m = len(l)//2 if l[m] == x: return True elif l[m] > x: return bin_search(x,l[:m]) return bin_search(x,l[m+1:])</pre>

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) & Remarques
Déclencher une exception	<pre>raise Exception("message") # ou bien raise Exception()</pre>	<pre>raise TypeError(" entier attendu ! ")</pre>
Prévoir une exception	<pre>if cond_erreur : raise Exception("message") ou bien assert not cond_erreur, " message "</pre>	
Intercepter une exception	<pre>try : bloc_risqué 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>

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

Description	Syntaxe	Exemple(s) & Remarques
Ouverture d'un fichier. L'ouverture d'un fichier permet de créer une connexion entre le programme Python et le fichier spécifié par un <code>str file_name</code>. Le deuxième paramètre mode est un <code>str</code> permettant d'indiquer le mode d'ouverture il peut avoir une des valeurs suivantes : Mode Lecture seule (<code>mode = 'r'</code>) : Le fichier doit exister et seule la lecture est autorisée. L'ouverture d'un fichier inexistant génère une erreur. Mode Écriture (<code>mode = 'w'</code>) : Si le fichier n'existe pas, il est créé, sinon il est écrasé. Mode mise à jour (<code>mode = 'a'</code>) : identique au mode 'w', sauf que si le fichier existe, il n'est pas écrasé et l'ajout se fait en fin du fichier.	<pre>f = open(file_path, mode)</pre>	— Ouverture du fichier a.txt en mode lecture <code>f = open("a.txt", "r")</code> — Ouverture de fichier a.txt en mode écriture <code>f = open("a.txt", "w")</code> — Ouverture en mode mise à jour <code>f = open("a.txt", "a")</code>
Lecture à partir d'un fichier texte.	Lecture intégrale : <pre>f.read()</pre> renvoie un str contenant le contenu intégral du fichier (à ne réserver qu'aux fichiers de taille raisonnable). Lecture Ligne par ligne : <pre>f.readline()</pre> Fournit la ligne courante sous forme d'un str et place le curseur sur la ligne suivante. Elle renvoie la chaîne vide si fin du fichier. Lecture intégrale ligne par ligne : <pre>f.readlines()</pre> Fournit la totalité du contenu du fichier sous forme d'une liste de str. Le caractère de saut de ligne '\n' présent à la fin de chaque chaîne indique la fin de chaque ligne. Itérer sur un fichier ligne par ligne : <pre>for line in f: #traitement sur une ligne de f</pre>	Soit le fichier noms.txt ayant le contenu suivant cem may up On suppose que les instructions de lecture sont toujours précédées par l'instruction d'ouverture <code>f = open("nom.txt", "r")</code> <code>f.read()</code> #renvoie le str "cem\nmay\nup" Le premier appel de <code>r.readline()</code> renvoie "cem\n" l'appel suivant renvoie "may\n" le 3e appel "up" les appels conséquents renvoient un str vide (car la fin de fichier est atteinte) L'appel <code>f.readlines()</code> renvoie la liste ["cem\n", "may\n", "up"] on obtient le même résultat en utilisant la compréhension [line for line in f]

Écriture dans un fichier Texte

Écriture d'un str :

```
f.write(ch)
```

Écriture d'une liste de str :

```
f.writelines(lst)
```

Fermeture d'un fichier. Cette instruction permet de fermer la connexion établie par open et de libérer les ressources réservées.

```
f.close()
```

L'ouverture du fichier précédent **nom.txt** avec l'instruction

```
f = open("noms.txt", "w")
```

va écraser les lignes déjà présentes dans le fichier. Ainsi, après l'ouverture le fichier **noms.txt** est à plat. noms.txt

Après l'instruction `f.write("cemcem\n")` le fichier aura le contenu suivant

cemcem

 L'instruction

`f.write("maymay\n")` permet d'ajouter une nouvelle ligne au fichier. Ainsi, son contenu évoluera comme

suit :

cemcem
maymay

 Il est possible d'obtenir le même effet avec l'instruction

```
f.writelines(["cemcem\n", "maymay\n"])
```

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.

Description	Exemple(s) & Remarques
Calcul du coût Le coût théorique se mesure en nombre d'instructions et se calcule comme suit : — Coût d'une opération élémentaire op = 1. — Coût d'une séquence de k instructions $p_1; p_2; \dots p_k = \sum_{i=1}^k \text{coût}(p_i)$. — Coût d'une structure conditionnelle simple if c: p else : $q \leq$ coût (c) + max(coût (q), 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) ⇒ Majorer le nombre de répétitions : NombreRépétitions*(coût(c)+ coût(p))+ coût(c)	<pre> — s += x + y # 3 instructions si x,y,s sont des instances de classes simples — if x > y : a += y # 1 + max(2,0) = 3 instructions — if x >= y and x >= z : s += x elif y >= z: s += y else: s += z # (3 + 1) conditions + 3 — for i in range(n): s += x**i # 2 + log_2(i+1) # 2 * n + sum(log(i) i=1..n) — while n != 0: #log_10(n + 1) répétitions s += n%10 #3 n //= 10 #2 #log_10(n) * (1 + 5) + 1 # 2 * n + sum(log(i) i=1..n) </pre>

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

Description	Exemple(s) & Remarques
Constante : $O(1)$	— Accès à un élément d'une liste <code>l[i]</code>. — Additionner multiplier un nombre constant d'objets.
Logarithmique : $O(\log(n))$	— Recherche dichotomique <code>n = len(lst)</code>. — 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 <code>n = len(lst)</code>. — Tester l'appartenance d'un élément dans une liste <code>n = len(lst)</code>. — Produit scalaire de deux vecteurs.
Quasilineaire : $O(n \times \log n)$	— Trier une liste <code>lst.sort()</code> : <code>n = len(lst)</code>. — Tri fusion / Tri rapide
Quadratique : $O(n^2)$	— Tri par sélection/insertion/bulles <code>n = len(lst)</code>. — 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