#2 Introduction au langage Python

Structures de données élémentaires :
variables, entiers, flottants, chaînes et listes

Édition 2021-2022

Plan

Bases du langage Python

Variables – Affectations, expressions, opérations et opérateurs et règles de priorité

Bases du langage : variables

Définition, généralités

- **Une variable est aussi appelée « conteneur » car elle contient une donnée**
- **Une variable est déclarée lors de sa première affectation.**
- **Le nom d'une variable, ou identifiant, est formé de caractères :**
  - Lettres (majuscules ou minuscules) non accentuées, des chiffres et le caractère soulignement « _ » (*underscore*) ; Python est sensible à la casse des lettres (ie MAJUSCULEs ou minuscules) ;
  - Les autres caractères spéciaux ne sont pas acceptés ;
  - Le premier caractère est toujours une lettre ou « _ » : **ne jamais commencer par un chiffre ;**
  - Utiliser de préférence des noms de variables illustrant leur contenu.

    Par exemple : `mon_age` **ou** `monAge` **plutôt que** `a` **ou** `ma`.
- **Les variables ne peuvent avoir comme identifiant un mot-clé du langage (Cf. diapositive suivante « mots clefs du langage »).**

Quelques exemples

- `1variable` **: non**
- `var_accentuée` **: non**
- `else` **: non (mot clef)**
- `VAR` **: oui mais éviter majuscules**
- **préférer :** `var`
- `Var` **: éviter (réservé classes)**
- `var32` **: oui**
- `v` **: éviter (pas « parlant »)** **préférer :** `variable` **ou** `var`
- `monidentifiant` **: oui**
- **Préférer :** `monIdentifiant` **ou** `mon_identifiant`
- **Éviter les abus comme** `CeciEstMaVariableDontLidentifiantEstBeaucoupTropLong`

Bases du langage : mots clefs du langage

Définition

- Les mots clefs (*keyword* en anglais) ou mots réservés d'un langage sont les instructions de base qui permettent de composer de nouvelles instructions.

Liste des mots clefs en Python

- Les mots-clés sont fournis dans la liste `keyword.kwlist` du module keyword (Cf. partie « Modules », situé après les « Fonctions » dans ce diaporama).
- Les mots-clés de Python 3.4 sont les suivants (enlevé en (bleu) pour V3.9) :
  - `and, as, assert, break, class, continue, def, del, elif, else, except, (exec), False, finally, for, from, global, if, import, in, is, lambda, none, nonlocal, not, or, pass, print, raise, return, True, try, while, with, yield.`
  - `Obtenu par :`

```
>>> Import keyword
>>> keyword.kwlist()
```

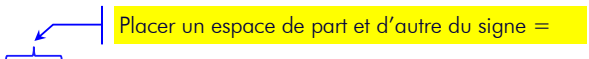
- Pour rappel, les variables ne peuvent avoir comme identifiant un mot-clé du langage

Bases du langage : variables et affectations (ou assignations)

Définitions

- L'affectation est l'action qui établit un lien entre le nom de la variable et son contenu
- Le symbole de l'affection en Python est « = » (« ← » en pseudo-langage « := » en Pascal).

Exemples



- `anneeConcours = 2022` : après affectation, le contenu de `anneeConcours` est `2022`.
- `monResultat = 2*3` : après évaluation, 6 est affecté à la variable « `monResultat` ».

Type (d'une variable)

- Une variable dispose d'un type attribué au moment de l'affectation : c'est un typage dynamique.
- Une variable change de type si elle est réaffectée (on dit « écraser » la variable).
- La notion de type sera développée un peu plus loin.

Commentaires

- Sur une ligne, le texte placé après le caractère « # » et jusqu'à la fin de la ligne, n'est par traité par l'interpréteur : c'est un commentaire utilisé pour documenter son script
- Pour rédiger un commentaires sur plusieurs, lacer le texte entre guillemets triple (`"""`) ou triple apostrophe (`'''`). Comme tou commentaire, ce texte n'est pas interprété.
- Exemples : `a = 12 # Commentaire`
  ou `b = a-2 ''' Commentaire…`
  `… sur plusieurs lignes '''`

Bases du langage : variantes sur les affectations

Réaffectation

- **Une variable déjà définie peut à nouveau être affectée : c'est une réaffectation**
  - **Exemple :** `a = 1 ; b = a ; a = 4 # a vaut 1, puis 4 et b vaut 1 au final`
  - **Autre exemple courant :** `p = 2 ; p = p+5 # p = 2 puis est remplacé par 7 (= 5+2)`

Affectations multiples

- **Possibilité d'affecter plusieurs variables sur une ligne en une seule fois chaînage des « = »**
  - **Exemple :** `a = b = c = 2` **# La valeur « 2 » est affectée simultanément à** `a`, `b` **et** `c`.

Affectations parallèles

- **Possibilité d'affecter des variables en les listant à gauche et leur affectations à droite**
  - **Exemple :** `a, b = 1, 2 ; # a prend la valeur 1 et b la valeur 2.`

Échange de contenus de deux variables

- **L'affectation croisée de variable échange les contenus**
  - **Exemple :** `x, y = 1,2 ; y, x = x, y # à terme x contient 2 et y cont. 1.`
- **Ne fonctionne que si les variables ont été affectées au préalable (sinon erreur).**

Incrémentation, décrémentation, multiplication, etc.

- **Exemples :** `i += 1 # équivalent à i = i+1` **et aussi :** `i -= 1 # comme i = i-1.`
- **Et :** `i *= 2 # i=i*2 ; i /= 7 # i=i/7` **ou** `i **= 3 # i=i**3` **(élévation à la puiss. 3).**

Bases du langage : expressions arithmétiques et opérateurs

Expressions arithmétiques et expressions logiques

- **Une expression arithmétique est une combinaison d'opérations utilisant les opérateurs arithmétiques du langage (voir le tableau ci-dessous à gauche)**
- **Une expression logique est une combinaison d'opérations utilisant les opérateurs logiques du langage (tableau ci-dessous à droite)**

Opérations et opérateurs

Opérateurs arithmétiques

Opérations arithmétiques	Symboles	Exemples
Addition	+	`2 + 5` renvoie `7`
Soustraction	–	`8 - 2` renvoie `6`
Multiplication	*	`6*7` renvoie `42`
Exponentiation (puissance)	**	`5 ** 3` renvoie `125`
Division	/	`7 / 2` renvoie `3.5`
Reste de division entière	%	`11 % 4` renvoie `3`
Quotient de division entière	//	`7 // 3` renvoie `2`

Opérateurs logiques

Opérations logiques	Symboles	Exemples
identité logique (oui)	—	—
complément logique (non)	`not` ou `!`	Not True renvoie `False`
et (and)	`and` ou `&`	`True and False` renvoie `False`
ou (or)	`or` ou | (altGr+6 / ⇑+alt+L)	`True or False` renvoie `True`
ou exclusif (xor)	^ (bit à bits)	`True ^ True` renvoie `False`
égalité / identité	`==` (égal) ou `is` / `!=` (différent)	`1==2` ou `1 is 2` renvoient `False`
inégalités	`< ; <= ; > ; >=`	`1>2` (`False`) ; `2<=2` (`True`).

Bases du langage : règles de priorités dans les expressions arithmétiques (1/2)

Règles de priorités des opérations dans les expressions

- **Il arrive souvent que plusieurs opérateurs soient employés dans une seule expression**
- **Deux cas se présentent dans l'évaluation des expressions :**
  1. Un ordre de priorité « naturel » est établit, régit par : les « règles de priorité des opérations » ;
  2. L'ordre est transgressé (modifié ou forcé) par l'usage de parenthèses
- **En Python, l'ordre des priorités qui s'applique est celui défini en mathématiques**

Rappel de l'ordre des opérations mathématiques

- **Un produit est prioritaire sur une addition (ou une soustraction)**
  - `1 + 2*3` **signifie : évaluation de** `2*3` **puis** `1` **est ajouté (résultat** `7`**). Et pas :** `1+2` **(*3) puis** `3*3=9`
- **Une élévation à une puissance est prioritaire sur une multiplication (ou la division), et par conséquent sur une addition :**
  - `2 * 3**2` **signifie : évaluation de** `3**2` **puis multiplication par** `2` **(résultat** `11`**).**
    Et pas : `2*3` **(**`**2`**) puis** `6**2=36`
  - **Ou alors, avec la notation compacte « e » (à privilégier) :** `2 * 3e2 = 600` **et pas** `2*3*10**2`
- **L'application d'une fonction à un argument est prioritaire ; en fait la mise entre parenthèses de l'argument évite cette situation :**
  - $\sin\frac{\pi}{2}$, **alors qu'en Python, nous écrirons toujours** `sin(pi/2)`**, ce qui évite les problèmes…**

Bases du langage : règles de priorités dans les expressions arithmétiques (2/2)

Un « truc » mnémotechnique, abrégé par le terme « PEMDAS », permet de mémoriser aisément l'ordre des priorités

- **P pour parenthèses qui impose la priorité maximale en forçant l'évaluation d'une expression dans un ordre imposé. C'est un outil « massue » à éviter quand cela est possible.**
  - **Exemples : justifié pour** `2*(3 – 1) = 4` **et** `(1 + 1)**(5 – 2) = 8` **; moins pour** `(2*3) – 4` **ou** `3*(1e(–3))`
- **E pour exposants. Les exposants sont évalués ensuite, avant les autres opérations**
  - **Exemples :** `2**2+1 = 5` **(et pas** `2**(2+1) = 8`**) ;**
    et `3*1**10 = 3` **(et pas** `(3*1)**10 = 59049` **!)**
- **M et D pour multiplication et division, qui ont la même priorité. Elles sont évaluées avant l'addition A et la soustraction S, qui sont donc toujours effectuées en dernier lieu**
  - **Exemples :** `2*3–1 = 5` **(et pas** `2*(3–1) = 4`**) ;**
    `2/3–1 = 1/3` **(**`0.3333…` **En mémoire) (plutôt que** `2/(3–1) = 1.0`**)**
- **Si deux opérateurs ont la même priorité, l'évaluation est effectuée de gauche à droite suivant l'ordre naturel d'évaluation. Attention donc aux pièges !**
  - **Exemple : dans l'expression** `59*100//60`**, la multiplication est effectuée en premier, et la machine doit donc ensuite effectuer** `5900//60`**, ce qui donne** `98`**. Si la division était effectuée en premier, le résultat serait** `59` **(pour rappel l'opérateur** `//` **effectue une division entière).**

Bases du langage Python

Types scalaires : booléen, entier, flottant et complexe, caractère

Types séquentiels : chaîne et liste

Changement de type

Objets scalaires : types booléen et entier

Type booléen (boolean)

Majuscule Majuscule

- **Variable ou expression ne prenant que les valeurs vraie (True) ou fausse (False).**
- **Pour connaître le type d'une variable, utiliser la fonction « type »**
  - **Ex. :** `v = True ; type(v)` **renvoie** `<class 'bool'>` (idem pour `type(False)`)
- **Un seul bit suffit pour représenter des variables codées dans ce type : « 0 » ou « 1 ».**

Type entier (int)

- **Les entiers de Python sont à considérer au sens d'entiers relatifs ($\in \mathbb{Z}$) : ils sont signés.**
- **Connaître le type d'un entier, c'est toujours la fonction « type » qui est employée**
  - **Exemple :** `a = 12 ; type(a)` **renvoie** `<class 'int'>`
- **Un entier en Python n'a pas de limite de taille, à la différente du langage C (par exemple) pour lequel la taille dépend de celle des registres du microprocesseur, souvent 32 ou 64 bits.**
  - **Mais attention : le module scientifique « Numpy » code les entiers sur 64 bits.**
- **Écriture dans différentes bases : 2 (binaire), 8 (octal), 10 (décimal) ou 16 (hexadécimal)**
  Exemples : `n = 165` **(en décimal)**

  `bin(n)` renvoie `'0b10100101'`, le chiffre 0 en tête confirme que c'est un nombre
  (le caractère ' est un délimiteur de chaîne, car la fonction « bin » renvoie des chaînes : Cf. plus loin)
  `hex(n)` **renvoie** `'0xa5'` (c'est aussi une chaîne)
  `oct(n)` **renvoie** `'0o245'` (idem en octal avec un « o » minuscule)

Type flottant (float) – Cf. https://fr.wikipedia.org/wiki/IEEE_754

- Suivant la norme IEEE 754, un flottant est compris entre $-1{,}7\times10^{308}$ et $+1{,}7\times10^{308}$
- Attention, le point « . » est le séparateur décimal (la virgule est un autre séparateur)
- Un flottant est représenté sous cette forme : $x = s \times m \times 2^e$
  - s est le signe, bit de poids fort de m (bit numéro 63)
  - m est la mantisse, codée sur 52 bits, $m \in [1,2[$ (bits entre les numéros 0 et 51)
  - e est l'exposant, entier, codé sur 11 bits, $e \in [-1022 = -(2^{10}-2), 1023 = 2^{10}-1]$ (bits compris entre les rangs 52 et 62)

$$x = s \times m \times 2^e$$

- Un flottant est donc codé sur 64 bits, soit 8 octets

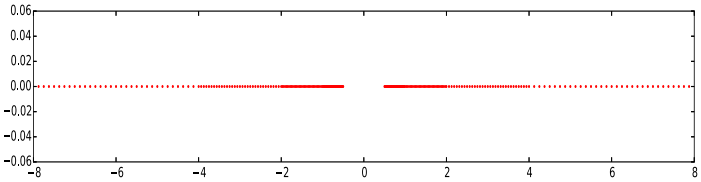
$$x = a_{63}a_{62}\dots a_{52}a_{51}a_{50}\dots a_1a_0$$

s signe — e exposant, $e = a_2 \dots a_{52}$ — m mantisse, $m = 1, a_0a_1 \dots a_{50}a_{51}$

Représentation non uniforme des flottants
Avec la représentation en « mantisse et exposant », la distance entre deux nombres flottants consécutifs est variable

Illustration ci dessous : représentation des flottants sur 8 bits

- Exemples
  - $5.1_{(10)} = 4+1+0{,}1 = (101.0001100011000\cdots)_{(2)}$
  - $0.1_{(10)} = (0.000110001100011\cdots)_{(2)}$
  - $2.125_{(10)} = 10.001_{(2)}$



Type complexe (complex)

- Comme en mathématiques, un complexe est l'association de deux réels, donc deux flottants
  - Exemple d'écriture : `z1 = complex(3,4) ; z2=1.5+0.5j`
- Usage de la fonction « type »
  - Exemple : `type(z1)` renvoie `<class 'complex'>`
- Partie réelle et imaginaire : `a.real ; a.imag`
  - Exemples : `z1.real` renvoie `1.5` ; `z2.imag` renvoie `4`
  - L'instruction qui suit le point (postfixée) est une méthode : elle agit sur la variable `z1` (ou `z2`)
- Opérations sur les complexes : c'est comme en maths
  - `z1+z2` renvoie `(4.5+4.5j)` `# Les parenthèses montrent que c'est un tout indissociable`
  - `z1-z2` renvoie `(1.5+3.5j)` `# Idem pour les parenthèses`
  - `z1*z2` renvoie `(2.5+7.5j)`
  - `Z1/z2` renvoie `(2.5999999999999996+1.7999999999999998j)`
  - `z1.conjugate()` renvoie `(3-4j)` `# Est-ce bien le conjugué de z1 ?`
- Remarques
  - `j` renvoie une erreur : `NameError: name 'j' is not defined#`
  - `1j` renvoie : `j` `# Correct ! Placer 1 devant j évite de confondre avec une variable`

Type caractère (char)

- **Type non numérique pour décrire du texte représenté dans des chaînes (Cf. ci-après)**
- **Les caractères sont codés en ASCII (*American Standard Code for Information Interchange*)**

Caractères de contrôle (00 à 31 + 127)

ASCII	Caract.	Signification	ASCII	Caract.	Signification
00	NUL	*null*, nul	16	DLE	*data link escape*, échap. liaison données
01	SOH	*start of heading*, début d'en-tête	17	DC1	*device control* 1, commande unité 1
02	STX	*start of text*, début de texte	18	DC2	*device control* 2, commande unité 2
03	ETX	*end of text*, fin de texte	19	DC3	*device control* 3, commande unité 3
04	EOT	*end of transmission*, fin de transmission	20	DC4	*device control* 4, commande unité 4
05	ENQ	*enquiry*, interrogation	21	NAK	*negative acknowledge*, acc. récep. nég.
06	ACK	*acknowledge*, accusé de réception	22	SYN	*synchronous idle*, inactif synchronisé
07	BEL	*bell*, sonnerie	23	ETB	*end of transmission block*, fin tran. bloc
08	BS	*backspace*, espacement arrière	24	CAN	*cancel*, annuler
09	HT	*horizontal tabulation*, tabulation horiz.	25	EM	*end of medium*, fin du support
10	LF	*line feed*, saut de ligne	26	SUB	*substitute*, substitut
11	VT	*vertical tabulation*, tabulation verticale	27	ESC	*escape*, échappement
12	FF	*form feed*, saut de page	28	FS	*file separator*, séparateur de fichiers
13	CR	*carriage return*, retour chariot	29	GS	*group separator*, sép. de groupes
14	SO	*shift out*, hors code	30	RS	*record separator*, sép. d'enregistr.
15	SI	*shift in*, en code	31	US	*unit separator*, séparateur d'unités
			127	DEL	*delete* (effacer, supprimer)

Caractères imprimables (32 à 126) ou table ASCII standard

ASCII	Caractère
32	SP (space, espace)
33	!
34	"
35	#
36	$
37	%
38	&
39	'
40	(
41	)
42	*
43	+
44	,
45	-
46	.
47	/
48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9
58	:
59	;
60	<
61	=
62	>
63	?

ASCII	Caractère
64	@
65	A
66	B
67	C
68	D
69	E
70	F
71	G
72	H
73	I
74	J
75	K
76	L
77	M
78	N
79	O
80	P
81	Q
82	R
83	S
84	T
85	U
86	V
87	W
88	X
89	Y
90	Z
91	[
92	\
93	]
94	^
95	

ASCII	Caractère
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r
115	s
116	t
117	u
118	v
119	w
120	x
121	y
122	z
123	{
124	|
125	}
126	~

Forçage du type lors de l'affectation

- **Un entier peut être forcé en type flottant en plaçant un point à droite lors de l'affectation**
  - Exemple : `a = 1 ; type(a)` renvoie `int` ; mais `a = 1. ; type(a)` renvoie `float`.
- **Le forçage peut est réalisé pendant les évaluations et les affectations**
  - Exemple : `2**3` renvoie 8 (opération entre entiers) ; `2.**3` renvoie `8.0` (c'est un flottant).

Changement de type avec affectation

- **Conversion en entier, avec la fonction `int(val)` qui renvoi un entier (`val` peut être un entier ou un flottant)**
  - Exemples : `int(1.)` renvoie `1` ; `a = int(-2.4) # a=-2` ; `int(136.5678e2)` renvoie `13656`
- **Conversion en flottant, avec la fonction `float(val)` qui renvoi un flottant**
  - `val` peut être un nombre entier ou flottant, et même une chaîne (compatible avec un nombre)
  - **Ex.** : `float(1)` renvoie `1.0` ; `a = float(-26//5) # a=-6` ; `float('123e-2')` renvoie `1.23`
- **Conversion en complexe :** `nb_complexe = complex(nb_flottant) # comme pour les entiers`
  - Exemples : `complex(2.5)` renvoie `(2.5+0j) # Car créer, c'est aussi convertir à partir de l'argument`
    les () encadrent les deux parties de complexe
- **Conversion d'un nombre en chaîne et d'une chaîne en nombre**
  - Exemples : `a = float('2.3') # a=2.3 ; b = int(a) # b=2 ; c = eval('3+4/2') # a=5.0`
  - Exemples : `str = str(a) # transforme la valeur 'a' en chaîne : a='5.0'`

L'essentiel sur ce vaste sujet

Type chaîne (string)

- Une chaîne est un assemblage de caractères, à l'instar des mots ou des phrases
- Les chaînes sont encadrés de guillemets simples ' (*quote*) ou doubles " (*double quote*)
  - Exemples : `'Chaîne1'` ou `"Chaîne2 »` sont équivalents
- Une apostrophe dans une chaîne est marquée par `\'` ou en imbriquant les signes
  - Exemples : `'l\'apostrophe'` ou `"l'apostrophe"`
- Certains caractères non imprimables (0<code ASCII<31) sont précédés de « \ »
  - Exemples : `'Passage à la ligne : \n'` ou `'\t pour une tabulation' (…)`
- Indiquer son rang dans une chaîne permet d'accéder à chaque caractère
  - Exemples : `str = 'ma chaîne' ; str[0] = 'm' ; str[2] = ' ' ;str[6] = 'î'` …
- Assemblage de chaînes par concaténation
  - Ex. : `str1 = 'ma ' ;str2 = 'chaîne' ;str =str1+str2` contient `'ma chaîne'`
- Détermination de la longueur d'une chaîne (son nombre de caractères) : fonction `len()`
  - Exemple : `str = 'ma chaîne' ; len(str)` renvoie la valeur `9`
- Répétition de chaînes (le bégaiement des chaînes)
  - Exemple : `str = 2*'cou'` renvoie `'coucou'` (forme particulière de concaténation)
- Le module « string » apporte des fonctions supplémentaires de gestion des chaînes

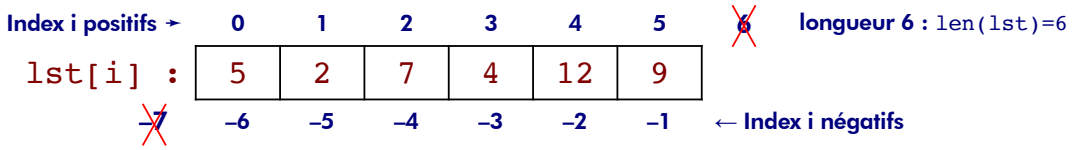
Définition

- Une liste est une suite d'objets séparés par une virgule, placée entre crochets ouvrant ([) et fermant (]). Une liste peut être hétérogène quand les objets sont de types différents (`str`, `float`, `int`, `complex`, `boolean`, etc.) et même des listes (notion de listes de listes).

Écriture de listes : vide, `list()` ou `[]` ou avec plusieurs éléments, `[0]*3` (=`[0,0,0]`)

Accès à un élément

- Comme pour les chaînes : `lst[i]` est le i-ème élément de la liste `lst` : `0 ≤ i < len(lst)`.
- L'indexation arrière est possible (index négatifs, conduisant à une indexation circulaire)
- Pour `lst = [5, 2, 7, 4, 12, 9]` (pour la lisibilité, placer une espace après la virgule)



Accès à une sous-liste ou tranche (technique du tranchage ou « slicing » an anglais)

- `lst[i:j]` désigne la sous-liste de `lst` formée des caractères d'indices `i` à `j-1` (`j` exclu).
- Si le début `i` n'est pas précisé, il est pris égal à 0 « par défaut » : `lst[:j] = lst[0:j]`.
- De même si `j` n'est pas précisé, il vaut `n = len(lst)`: `lst[i:] = lst[i:n]`.
- Indication d'un pas diffèrent de 1 : `lst[i:j:pas]` désigne la sous-liste de `lst` formée des éléments d'indices `i` à `i+pas*k` avec `j-pas*k ≤ i+pas*k < j`)

Objets séquentiels : type liste (2/2)

Exemple

```
>>> lst = [14, float(3), 'chaîne'] # Pour la lisibilité, espace après la virgule
>>> len(lst)
3
```

Réaffectation

- Contrairement aux chaînes, les listes sont des séquences modifiables. Un éléments peut être remplacé par simple affectation. La liste d'origine est donc modifiée.

```
>>> liste = [1, 2, 3, 4, 5]
>>> liste[1] = 4 # L'élément d'indice 1, c'est-à-dire le deuxième, est modifié.
>>> print(liste)
[1, 4, 3, 4, 5]
```

Très utilisé : ajout d'élément en fin de liste

- Usage de la méthode `append()` (détails sur les méthodes plus loin dans ce document)
- Attention ! Cette méthode modifie le contenu de la liste, ne renvoie rien et ne peut être affectée

```
>>> liste = [1, 2, 3, 4, 5]
>>> liste.append(6) # ajoute l'élément 6 à la fin de la liste
>>> print(liste)
[1, 2, 3, 4, 5, 6] # liste est modifiée (un élément en plus)
```

Concaténation de listes – Concaténation ≡ mise bout à bout

- Utiliser l'opérateur « + »

```
>>> lst1, lst2 = [1, 2, 3],[4, 5]
>>> lst1 + lst2
[1, 2, 3, 4, 5]
```


Pour voir plus loin : évocations d'autres types (développement dans C5)

D'autres types de variables existent, en particulier les types structurés

Ces types feront l'objet d'un cours et d'études particulières

Les types de données « itérables »

- Les t-uples ou « tuple », ou n-uplet, sont des listes immuables (c'est-à-dire non modifiables après création) d'objets hétérogènes (qui n'ont pas la même structure, donc pas le même type).
- Les « list » (listes) sont des tableaux dynamiques de données hétérogènes, de contenu et de taille modifiables.

Le type « tableau » (array)

- Extension de la notion de listes de listes, qui est propre aux modules scientifiques , tel que Numpy.
- Mathématiquement, les tableaux s'apparentent aux matrices en dimension 2 et aux tenseurs en dimensions supérieures.
- Les dimensions 2 (plan) et 3 (espace) sont les plus courantes (au-delà c'est plus abstrait).

Le type « fichier » (file)

- Ce sont des données scalaires ou elles-mêmes structurées (hors fichiers) stockées sur un périphérique de stockage (disque dur, SSD, clef USB), mais pas en mémoire.

A suivre donc, dans un autre document…

Références

Sur le net et ailleurs (si si, c'est possible…)

Références

Livres

- Apprenez à programmer en Python, Vincent Le Goff, Openclassrooms, 2ème édition, 2014
- Le cours développé dans ce livre est disponible en ligne sur le site web https://openclassrooms.com/courses/apprenez-a-programmer-en-python

Wikipédia

- https://fr.wikipedia.org/wiki/Python_(langage)
- https://fr.wikipedia.org/wiki/IEEE_754

Cours et références en ligne

- https://www.python.org
- https://www.developpez.com et https://python.developpez.com
- http://desaintar.free.fr/ (site personnel de Arnaud de Saint Julien)

Autres cours et références

- Documents de cours d'informatique de Sophie Dion, lycée Châtelet, Douai
- Cours d'informatique de Xavier PESSOLES et Damien ICETA, UPSTI

Fin de « Introduction au langage Python »

1 – Variables, nombres et chaînes

Avez-vous des (d'autres) questions ?