Informatique – Ressources pour les cours

À CONSERVER ET APPORTER À CHAQUE ACTIVITÉ D'INFORMATIQUE

#0

Découverte et prise en main du langage Python

Démarrage et aperçus généraux

Édition 2024-2025

Plan

Objectif de ce document – Repères de progression

En quoi consiste ce premier document (fourni sous forme de diaporama) ?

- Introduire les premiers éléments du langage Python : types de données et les structures
- Les détails ne sont pas présentés : ils feront l'objet développements spécifiques

Découpage des connaissances abordées

- Le langage est traité en cinq axes mis en évidence par l'algorithmique
  - Ce qui diffère, c'est la syntaxe et la précision nécessaire pour un langage
- Axe 1 : les extensions du langage : fonctions, modules et méthodes
- Axe 2 : les structures de données de base du langage Python (variables et types)
- Axe 3 : les alternatives (structures conditionnelles)
- Axe 4 : les itérations (structures itératives ou boucles)
- Axe 5 : les séquences, avec les chaînes, les listes et les n-uplets (tuples)

Raisons et intérêt de cette « première passe »

- Structurer et approfondir l'apprentissage du langage Python abordé en travaux pratiques
- Illustrations des notions par des exemples et des applications
- Laisser à plus tard les détails, la richesse et la souplesse de ce langage (diaporama 1 à 5)

Renvoi aux diaporamas détaillés : axes 1 à 5 = [C]ours de 1 à 5

- Localisation des informations détaillées : [Cn/Dm] avec n = N° diaporama et m = N° diapositive

Éléments pour favoriser l'apprentissage du langage Python (et tout autre langage)

Quelques conseils durant l'exposé des cours

- Chaque définition et chaque notion du langage met en œuvre des exemples illustratifs
- Durant leur présentation, prendre des notes pour compléter ce document support
  - Ajouter des informations supplémentaires ou d'autres exemples issus du contexte du cours (réponses aux questions des auditeurs, nouveaux exemples, des idées, etc.)
- Des exemples applicatifs illustrent les différentes notions abordées
  - Ils reprennent les exemples développés dans le TP1 « découverte du langage Python »
  - Ils ne sont pas rappelés dans ce document ; il faudra s'y référer au besoin

Rien de mieux que la pratique pour assimiler les notions nouvelles d'un langage

- Favoriser la pratique en « se mettant devant l'ordinateur »
  - Utiliser un EDI de son choix (EduPython, Anaconda, Thonny ou autre, suivant ses goûts)
  - Faire toutes les manipulations et essais, sans limite, pour bien comprendre « ce qu'il se passe » et comment l'interface de l'EDI renvoie les informations du langage
- Essayer les scripts proposés dans les diaporamas pour mieux étudier le cours
  - La mise en situation est préférable pour établir et renforcer les liens entre connaissances
  - Testez, essayez les scripts et étendez votre expérience…

Introduction

Langage(s) informatique(s) et historique de Python

Introduction (1/2)

Qu'est-ce qu'un langage de programmation informatique ?

- Un langage de programmation est un ensemble des notations conventionnelles destinées à formuler des algorithmes pour produire des programmes exécutables qui, en les exécutant, mettent en œuvre les algorithmes.

Un langage informatique a recours aux mêmes composantes qu'un langage naturel :

- Il est composé d'un alphabet (lettres latines EN-US le plus souvent) ;
- D'un vocabulaire composé des mots-clefs ou instructions ;
- De règles de grammaire appelées « syntaxe »
- Pour atteindre une signification qui prends corps dans l'usage du langage.

Un langage apparaît au travers d'une suite d'instructions dénommées lignes de codes

- L'ensemble des lignes de code constitue un « script » ou « code source » ;
- Le script est créé et modifié avec un éditeur (de texte) ;
- Cet éditeur est associé à d'autres outils de développement dans un environnement de développement intégré (EDI)
  - Pour Python, c'est par exemple : EduPython, Anaconda ou Thonny (il en existe plusieurs autres) ;
- Les scripts sont enregistrés dans un fichier de type « texte »
  - Pour Python, c'est par exemple : pour Python, avec l'extention « `.py` ».

Introduction (2/2)

En général, deux modes de mise en œuvre du code source coexistent

Traduction de la totalité du script du programme pour réaliser un exécutable

- **Cette opération est la compilation. Elle est réalisée par un programme appelé compilateur**
- **Pour un langage de programmation donné, l'opération se décompose en deux étapes**
  1. Transformer chaque ligne d'instruction lue dans le code source en un **code intermédiaire** appelé **code objet** ;
  2. Le code objet est combiné avec d'autres codes objet (provenant d'autres langages éventuellement) pour produire un code en **langage machine** composé de codes d'instructions que seul le processeur de l'ordinateur d'acceuil est en mesure d'exécuter
  - Pour information : si le code est exécuté sur l'ordinateur où a lieu la compilation, on parle de compilation simple ; si la compilation est destinée à une autre machine (avec un autre processeur, donc un autre jeu d'instructions), la compilation est dite « croisée » (cross-compilation en anglais). C'est le cas pour le développement de logiciels pour des machines de tout type, des « objets connectés » ou des smartphones.

Traduction au coup par coup de chaque ligne de code ou instruction

- **Cette opération est l'interprétation, réalisée par un programme appelé interpréteur**
- **Il** réalise **la traduction de chaque instruction en code machine « à la volée », puis ce code est transmis au processeur qui l'exécute et le cycle recommence pour chaque instruction suivante.**

Bref historique de Python

Développé vers la fin des années 80, première version (0.9) du langage en février 1991

Son auteur, Guido Van Rossum, travaille alors au *Centrum voor Wiskunde en Informatica* (CWI , Pays-Bas) dans lequel le langage ABC avait déjà développé : Python s'en inspire

Il nomme son langage « Python » car il est fan de la série « Monthy Python's Flying Circus »

Site web : https://www.python.org (documentation en ligne, en HTML et en PDF)

Et sur wikipedia : https://fr.wikipedia.org/wiki/Python_(langage)

Pourquoi utiliser le langage Python ? Ses atouts

- **Langage « puissant », pédagogique et riche de possibilités ;**
- **Disponibilité de larges bibliothèques prêtes à l'emploi (Cf. site) ;**
- **Possibilité de créer de nouvelles bibliothèques personnalisées ;**
- **Langage interprété : pas d'étapes fastidieuses de compilation ;**
- **Licence d'usage libre (non propriétaire) : chacun peut l'employer.**

Grandes réalisations en Python : Libre Office, Inkscape et chez Google

Deux versions majeures actives (octobre 2024)

- **Python 3.13 (3.12, octobre 2023) et Python 2.7.18 (2020-04)**

EduPython au lycée est en version 3.4 **Pas en CPGE**


Guido Van Rossum 2014

Bases du langage Python

1. Premiers éléments sur les structures de données : les variables, affectations, expressions, opérations et opérateurs et règles de priorité

Bases du langage : variables [C2/D7]

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 [C2/D8]

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 (en (bleu), ceux enlevés après 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) [C2/D9]

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

Pour une meilleure lisibilité : placer un espace de part et d'autre du signe =

- `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 [C2/D10]

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 [C2/D11]

Expressions arithmétiques et expressions logiques

- Une expression arithmétique est une combinaison d'opérations utilisant les opérateurs arithmétiques du langage (Cf. tableau ci-dessous, à gauche)
- Une expression logique est une combinaison d'opérations utilisant les opérateurs logiques du langage (Cf. 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).

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 Python

2. Structures de données :

détails sur les types de base, scalaires et séquentiels

Objets scalaires : types booléen et entier [C2/D12]

Type booléen (boolean)

- Variable ou expression qui ne prend que les valeurs vraie (True) ou fausse (False).
  (Majuscule) (Majuscule)
- 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 de ce type : c'est « `0` » ou « `1` ».

Type entier (int)

- Les entiers de Python sont à considérer au sens d'entiers relatifs ($\in \mathbb{Z}$) : ils sont signés.
- Pour connaître le type d'un entier, c'est toujours la fonction « type » qui est employée
  - Exemple : `a = 12 ; type(a)` renvoie `<class 'int'>`
- Les entiers n'ont pas de limite de taille en Python, à la différente du langage C par exemple où 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, 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)

Objets scalaires : type flottant [C2/D13]

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

$$x = a_{63}\underbrace{a_{62}\ldots a_{52}}_{e}\underbrace{a_{51}a_{50}\ldots a_1a_0}_{m}$$

s signe — e exposant, $e = a_{62}\ldots a_{52}$ — m mantisse, $m = 1,a_{51}a_{50}\ldots a_1a_0$

- Exemples
  - $0.1_{(10)} = (0.000110001100011\cdots)_{(2)}$
  - $5.1_{(10)} = 4+1+0{,}1 = (101.000110001100011\cdots)_{(2)}$
  - $2.125_{(10)} = 10.001_{(2)}$

Objets scalaires : type caractère [C2/D15]

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	~

Changement de type des variables scalaires

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)` ➔ `float`.
- **Le forçage peut est réalisé pendant les évaluations et les affectations**
  - Exemple : `2**3` ➔ `8` (opération entre entiers) ; `2.**3` ➔ `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.)` ➔ `1` ; `a = int(-2.4)` : `a` contient `-2` ; `int(136.567e2)` ➔ `13656`
- **Conversion en flottant, avec la fonction `float(val)` qui renvoi un flottant**
  - `val` peut être un entier ou flottant, et même une chaîne (compatible avec un nombre)
  - **Ex.** : `float(1)` ➔ `1.0` ; `a = float(-26//5)` : `a` contient `-6.` ; `float('123e-2')` ➔ `1.23`
- **Conversion en complexe :** `nb_complexe = complex(nb_flottant) # comme pour les entiers`
  - Exemples : `complex(2.5)` ➔ `(2.5+0j)` `# Car créer, c'est aussi …`
    les () encadrent les deux parties de complexe … `convertir à partir de l'argument`
- **Conversion d'un nombre en chaîne et d'une chaîne en nombre**
  - Exemples : `a = float('2.3') ; b = int(a)` (2 dans b) ; `c = eval('3+4/2')` (5.0 dans c)
  - Autre ex. : `str = str(a) # transforme la valeur de a en chaîne : '2.3 ' dans str`

Objets séquentiels : type chaîne (1/2) [C5/D4à10]

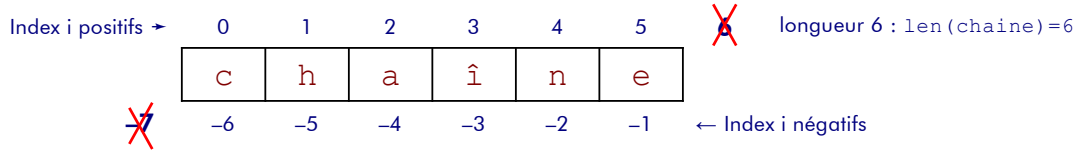
Définition

- **Une chaîne est un assemblage homogène de caractères permettant de coder des textes délimitées par des apostrophes (`'chaîne'`) ou guillemets (`"chaîne"`).**

Écriture de la chaîne vide : `''`

Accès à un élément (caractère)

- **`chaine[i]` est le i-ème caractère de la chaîne nommée `chaine`.**
- **L'indexation commence toujours à `0` : c'est pourquoi le dernier indice est `n-1`.**
- **L'indexation arrière est possible avec des index négatifs (l'indexation est « circulaire ») :**
  - **`chaine[-1]` est le dernier caractère de la chaîne `chaine` : ici `'e'`**



- **La chaîne, comme ses éléments (caractères), sont de même type, `str` :**
  - Exemple : `type(chaine[0])` renvoit `str`

Assignation (c'est-à-dire « modification ») d'un caractère d'une chaîne

- **L'accès à une chaîne possible en « lecture », mais « l'écriture », c'est-à-dire la modification d'un caractère de la chaîne est impossible : `chaine[0] = 'C'` renvoie cette erreur : `('str' object does not support item assignment)`**

Objets séquentiels : type chaîne (2/2) [C5/D4à10]

Détails pour le type chaîne (string)

- Pour placer une apostrophe dans une chaîne, on la marque par \' (`'\'` = caractère d'échappement) ou les marqueurs de chaînes sont imbriqués
  - 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
  - Ex. : `str = 'ma chaîne';str[0] = 'm';str[2] = ' ';str[6] = 'î';str[8] = 'e'`
- 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'` (c'est une forme particulière de concaténation)
- Le module « string » apporte des fonctions supplémentaires de gestion des chaînes
  - `import string as str` par exemple, puis consulter l'aide : `help('string')`

Objets séquentiels : type liste (1/2) [C5/D12]

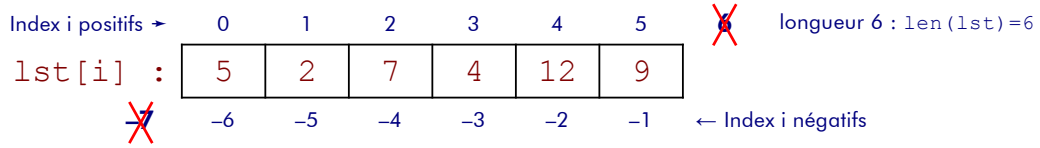
Définition

- 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 différent 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) [C5/D13à19]

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]
```


Objets séquentiels : type n-uplet ou tuples (et, « un peu », ensemble) [C5/D20à23]

Définition d'un n-uplet

- Un n-uplet ou tuple est une suite d'objets immuables (pas de réaffectation) séparés par une virgule, souvent placés entre parenthèses. Ils sont hétérogènes comme les listes.

Le tuple vide s'écrit : (). Un singleton doit être déclaré avec une virgule : (a,)

Un tuple est immuable

- On ne peut ni ajouter, ni retrancher, ni modifier des éléments d'un tuple : immuables dit-on
- L'accès aux éléments a lieu par indexation, comme pour les chaînes ou les listes.

Exemple : `a = (1, '3', [2,6,9], 'abc')`

`a[2]` renvoie `[2,6,9]`

Définition d'un ensemble (type qui apparaît au programme en 2021)

- Un ensemble (type `set`) est une collection d'objets non ordonnés séparés par une virgule et encadrés par des accolades (« { » et « } »). Comme pour les listes, ces objets peuvent être de types différents (`str`, `float`, `int`, `complex`, `bool`, etc.) : hétérogénéité.

L'ensemble vide s'écrit : {}.

Conversion : fonction `set()`

- Un objet de la classe `tuple`, `list`, `str` peut être converti en ensemble avec la fonction `set`.
- Les opérations sur les ensembles (union, intersection, etc.), les tests (appartenance, inclusion, etc.), les ajouts et suppressions d'éléments utilisent les méthodes `add` et `remove`. (détails dans la documentation Python).

Extension du langage Python

Les fonctions, les modules et les méthodes

Fonctions – Définitions [C1/D4]

Une fonction, qu'est-ce que c'est ?

- **Une fonction permet d'étendre les possibilités d'un langage par rapport à ses instructions élémentaires disponibles sans ajout particulier.**
- **Syntaxe de l'utilisation d'une fonction en Python pour récupérer le résultat qu'elle établit :**

```
resultat = nom_fonction(argument1, argument2,…, argumentN)
```

- **Trois éléments caractérisent une fonction :**
  - L'identifiant de la fonction `nom_fonction` est obligatoirement différents des mots réservés du langage et utilise les même règles que le nommage des variables. Éviter aussi de choisir le nom d'une variable déjà employée **(erreur en cas de référence à la fonction prise pour la variable)**
  - Les variables passées à la fonction, les `arguments`, sont placées entre parenthèses et séparés par des virgules (virgule = séparateur) : c'est la « matière d'œuvre » de la fonction sur lesquels elle agit. L'espace n'est pas obligatoire, mais peut améliorer la lisibilité
  - La variable `resultat`, un **scalaire** ou un **itérable**, reçoit les évaluations de la fonction (pour rappel, un **itérable** est souvent une liste ou un tuple, nous y reviendrons)
- **Parmi ces trois éléments :**
  - Le nom est obligatoire ;
  - Les arguments peuvent être absents, alors seules demeurent les parenthèses ;
  - Le résultat peut être optionnel (quand il n'y a rien à restituer) ;
  - Ces deux derniers cas dépendent des traitement effectués par la fonction

Fonctions – Exemples de fonctions prédéfinies [C1/D5]

Des fonctions préféfinies ont déjà été utilisées (`int`, `float`, etc.) ; en voici d'autres

Fonction `print()` pour afficher sur l'écran (à la position courante du curseur)

- **`print('txt1',var1,'txt2',var2,…)` : affiche des chaînes (`'xx'`) et/ou des conteneurs (dont les variables de différents types)**
- **Exemple :** `nbJours = 7 ; print('la semaine compte', nbJours, 'jours.')`
  renvoie sur la console : `la semaine compte 7 jours.`
- **Cette fonction a un nom (`print`), reçoit des paramètres en nombre quelconque, mais ne renvoie aucun résultat qui serait affecté à une variable (puisque c'est un échos sur l'écran)**
- **Signe d'élégance, remarquer l'espace ajouté automatiquement avant et après la variable.**

Fonction `input()` pour obtenir des informations à partir du clavier

- **`var_in = input('txt')` : acquisition au clavier d'une ou plusieurs variables qui seront renvoyées pour être affectées à** `var_in`
- **Exemple 1 (première manière de faire)**

```
print('Veuillez entrer un nombre entier')
n = input()
print('Le nombre que vous avez choisi est', n)
```

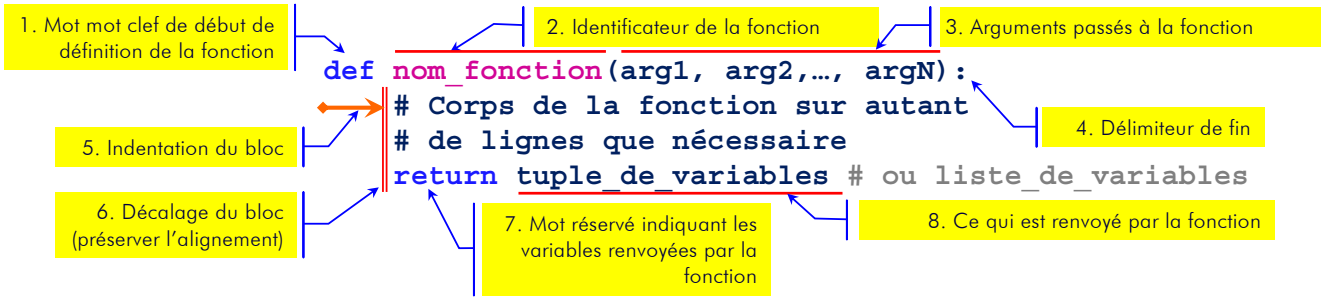
- **Exemple 2 (procédons autrement)**

```
n = input('Veuillez entrer un nombre entier\n') # Sans gestion d'espace
print('Le nombre que vous avez choisi est', n)
```

- **Attention ! La fonction `input` renvoie une chaîne : la convertir pour obtenir un nombre**

Fonctions – Fonction créée par le programmeur (1/3) [C1/D6]

Pour créer une fonction, il faut la déclarer : syntaxe de la déclaration d'une fonction



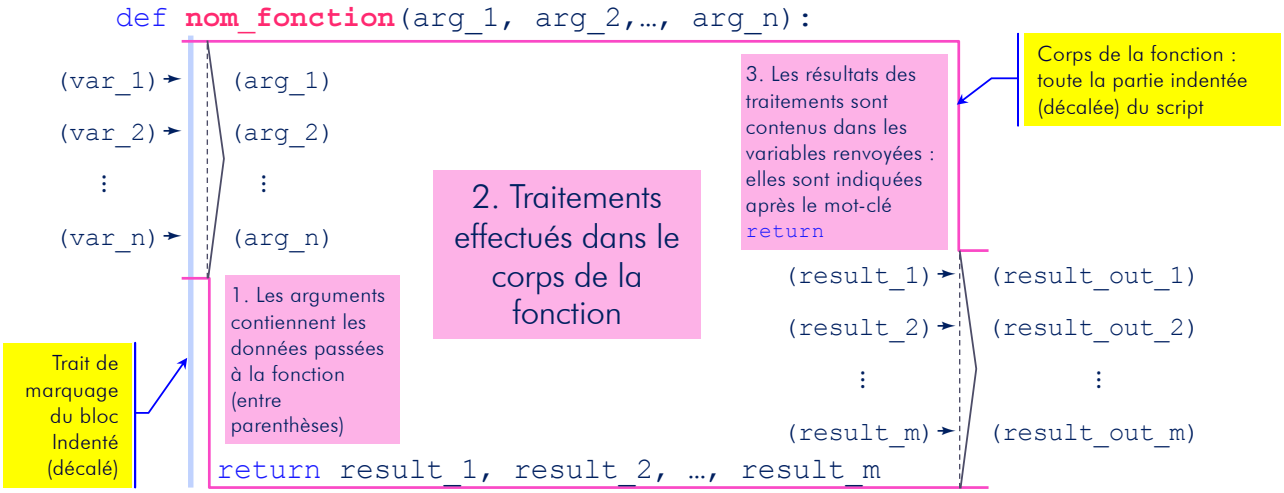
Usage de la fonction

- **Comme pour les fonctions prédéfinies : les règles d'usage sont strictement les mêmes**

Remarques

- **Il est possible qu'une fonction ne reçoivent aucun argument**
  - **Exemple :** `input() # Attend simplement d'entrer une valeur (sans message)`
- **Il est possible qu'une fonction ne renvoie aucun résultat à affecter à une variable**
  - **Exemple :** `print('Hello!') # Affiche Hello! et c'est tout`
- `Return` peut être suivi d'évaluations, ex.: **`return a, b, c`**
- Parfois, le corps de la fonction ne comporte qu'une seule ligne ; ex. : `return expression`

Mécanisme de passage des arguments et de renvoie des résultats



- `(x)` signifie « contenu de `x` ».
- Le contenu de chacune des variables `var_i` indiquées lors de l'appel de la fonction (variables dans l'espace global) remplace le contenu des variables argument `arg_i` utilisés dans le corps de la fonction (variables dans l'espace local).
- Le contenu de chacune des variables `result_j` (variables dans l'espace local) renvoyées par la fonction après le mot clé `return` remplace le contenu de chacune des variables `result_out_j` présentes dans l'espace global (« `out` » pour indiquer qu'elles sont extérieures à la fonction).

Exemple de création d'une fonction

- **Fonction `rectangle()` calculant, affichant et renvoyant l'aire et le périmètre d'un rectangle donné par la longueur de ses cotés (en arguments)**

- **Étape 1 - Script de définition de la fonction (un exemple possible)**

```
def rectangle(a,b):
    l = (a+b)*2
    s = a*b
    return s, l
```

- **Étape 2 - La partie du script permettant d'appeler la fonction pour l'utiliser**

```
aire, long = rectangle(a,b)
print("Aire et périmètre du rectangle :", aire, "et", long)
```

Modules [C1/D14&15]

Les modules permettent d'étendre les possibilités du langage en ajoutant des fonctions

- Un module est un bloc de code définissant des fonctions et des variables
- Il est enregistré dans un fichier dont le nom est celui du module qu'il faut « importer »
- Les fonctions et variables du module se regroupent autour d'usages communs

Un module peut être importé de trois manières différentes, en voici une pour l'instant

- Importer toutes les fonctions du module
- et lui associer un « alias » : `import nom_module_qui_est_long [as nomCourt]`
  - Un alias **([]≡optionnel)** permet de changer le nom du module en une abréviation plus concise
  - Le nom de la fonction est **préfixé** par `nom_module` **ou** `alias` : le séparateur est le point « . »
  - C'est plus long à écrire, mais le module nécessaire est connu : c'est plus sûr et auto-documenté
  - Exemple :

```
import math as m # importation de toutes les fonctions du module math
x = -m.pi/3 # affectation d'une valeur
valFonction = 4*m.sin(m.pi/4) + m.log(abs(x)) # calcul de la valeur
print(valFonction)
```

Pour connaître les fonctions contenues dans un module, taper :

```
>>> help('nom_du_module')
```


Exemple : autres aspects du module math [C1/D16]

Le module « math » ajoute à Python des fonctions mathématiques

- Importation : `import math`
- Les constantes : `.pi` et `.e` (pour rappel, le point précède la variable/fonction : `math.xxx`)

```
>>> math.pi
3.141592653589793
```

- Les arrondis et les troncatures : .ceil (supérieur), .floor (inférieur), .trunc (troncature)

```
>>>a = 1.23 ; math.ceil(a), math.floor(a), math.trunc(a)
(2, 1, 1)
```

- Puissance, racine : .pow, .sqrt

```
>>>b = 3 ; math.pow(b,3) , math.sqrt(b)
(27.0, 1.7320508075688772)
```

- Trigonométriques : .cos, .sin, .tan, .acos, .asin, .atan, .atan2, .cosh, .sinh, .tanh, .acosh, .asinh, .atanh
- Logarithmiques, exponentielles : .log, .log10, .log1p, .ldexp, .exp, .expm1, .frexp
- Conversion d'angles : .degrees, .radians

```
>>> alpha = math.pi/3 ; math.degrees(alpha) , math.radians(30)
(59.99999999999999, 0.5235987755982988)
```

- Gamma : .gamma, .lgamma
- Autres : `.fabs` (val abs), `.factorial`, `.fsum` (somme iterable), `.hypot` (distance euclidienne), `.fmod` (modulo), `.modf`, `.erf`, `.erfc`, `.copysign`
- test : `.isfinite` (est fini), `.isinf` (est infini), `.isnan` (nan=not a number)

Introduction

- Les fonctions reçoivent des arguments (entre parenthèses) et renvoient des résultats dans des variables (par affectation). Si la fonction ne renvoie rien, c'est noté `'None'`;
- Autre manière de faire : employer des méthodes. Elles évaluent les résultats en utilisant une syntaxe différente des fonctions : `variable.methode(…)`. Des détails…

Préliminaire : généralisation de la notion de variable

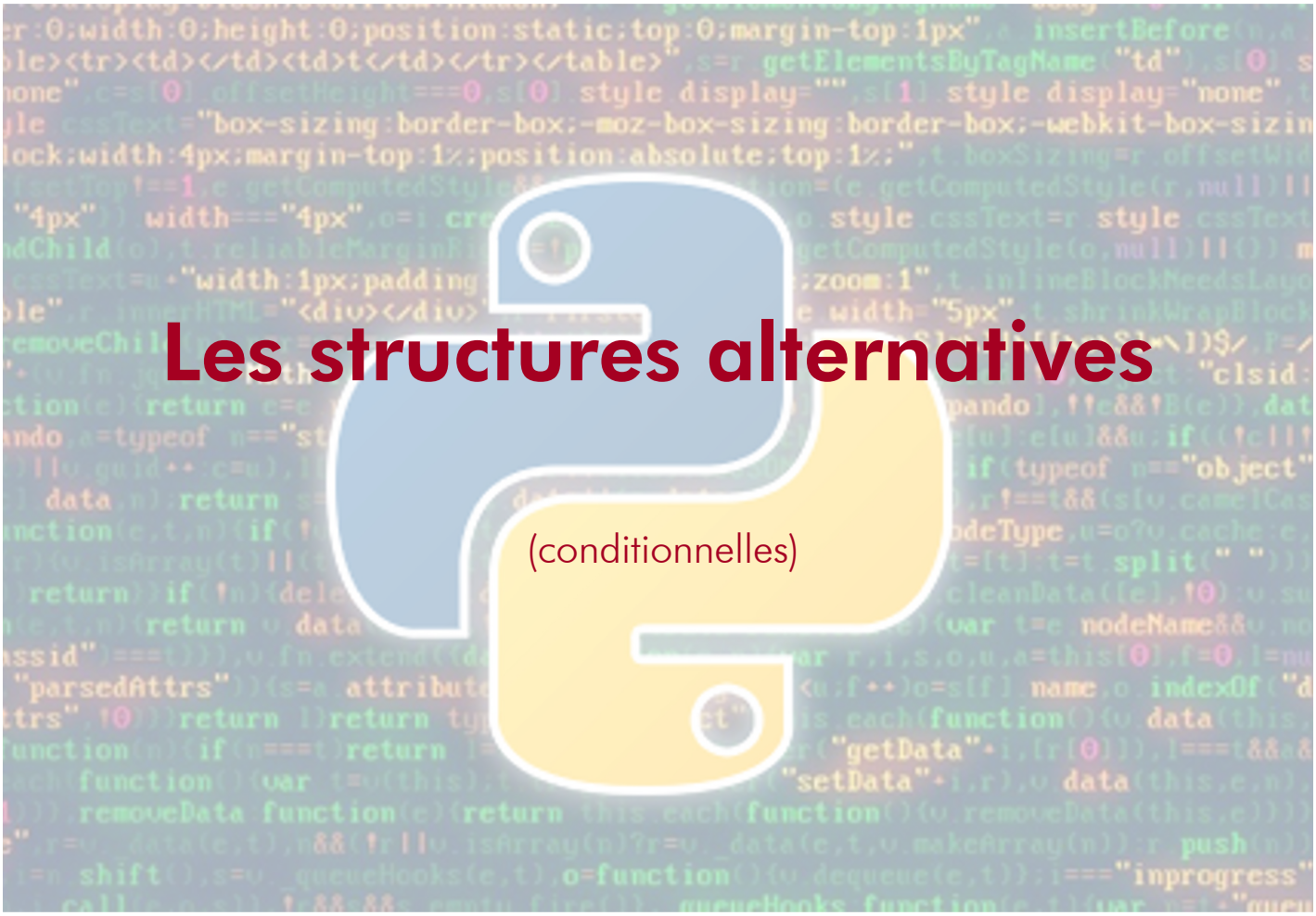
- Une variable en Python, c'est plus qu'une simple « boîte » dans laquelle se trouve une valeur : c'est en fait une conteneur appelé « objet ».
- Un objet est une structure de données contenant à la fois des données (scalaires ou structurées), mais aussi des fonctions en mesure d'agir sur les données de l'objet.
- Ces objets sont construits suivant des modèles appelés « classes » (des sortes de moules à objets)
  - C'est la raison pour laquelle la fonction « `type(var)` » renvoie le nom de la classe de `var`.

Définition d'une méthode

- Le terme « méthode » désigne les fonctions incluses dans la classe d'un objet.
- Ainsi une méthode s'applique aussi bien à une variable, qu'une fonction, etc.
- Mais comme la fonction est « incluse » dans la variable (pour rappel, c'est une classe),
- la variable est dabord indiquée avec à sa suite la méthode en séparant par un point.
- La syntaxe lors de l'utilisation d'une méthode est la suivante :

```
nom_variable.nom_methode(arg1,…, argN)
```

- Il est possible de ne pas passer d'argument : parenthèses vides (à voir dans les exemples)



Les structures alternatives

(conditionnelles)

Définition et différents types de structures conditionnelles (ou alternatives) [C3/D4à11]

Définition du vocabulaire relatif aux structures alternatives

- **Se sont les structures de contrôle qui permettent d'effectuer une ou plusieurs instructions en fonction de la réponse, vraie ou fausse, à une condition (qui est un prédicat)**

Suivant le nombre de cas répondant à la condition, il y a différentes alternatives

- **Alternative simple**
  - Une seule prise de décision : alternative « `if` »
  - Elle permet d'effectuer une ou plusieurs instructions si (si ≡ *if*) une condition est vérifiée.
- **Alternative complète**
  - Deux prises de décision : alternative « `if … else` »
  - Elle permet d'effectuer une ou plusieurs instructions si une condition est vérifiée, et une ou plusieurs autres instructions dans le cas contraire (sinon ≡ *else*).
- **Alternative de sélection**
  - Un nombre variable de prises de décision : alternative « `if … elif … else` »
  - Prolonge l'alternative complète en complétant la structure « `if … else` » avec des cas intermédiaires du type « sinon, si … » en cascade, ce qui contracte « `else-if` » en « `elif` »
  - Ici, dès que l'une des conditions intermédiaires est vérifiée, les instructions correspondantes sont effectuées, et ce qui suit est ignoré. Dans ce type de structure, un seul bloc d'instructions au plus est exécuté.

La condition est un prédicat qui prend une valeur vraie (`True`) ou fausse (`False`)

- `A = 2 ; a <= 0` **vaut** `False`
- `a = 2 ; b = -a/2 ; a%2 == 0 and b < 0` **vaut** `True`

Syntaxe complète des 3 alternatives : « if … », « if … else … » et «if … elif … else … »

Alternative simple

```
if <condition>:                 # Entête de l'alternative, se termine par " :"
      <Début du bloc traité>    # si <condition> est vrai
      <Suite du bloc>           # Le bloc est indenté…
      <Fin du bloc>             # sur plusieurs lignes.
<Instructions suivantes>        # rupture d'indentation = fin de bloc
```

Alternative complète

```
if <condition>:           # Entête
      <Bloc traité>       # si <condition> est vraie
else:                     # else ≡ sinon en anglais
      <Bloc traité>       # si <condition> est fausse
<Instructions suivantes>  # rupture d'indentation = fin de bloc
```

En résumé, quelques remarques

- Un seul « **if** » par alternative ;
- Un seul « **else** » au plus par alternative ;
- Autant de « **elif** » que l'on veut par alternative, mais pas forcement de « **else** »

Sélection

```
if <condition1>:          # Entête
      <Bloc traité>       # si <condition1> est vraie
elif <condition2>:        # elif ≡ else if : c'est une sélection,
    <Bloc traité>         # si <condition2> vraie (et <condition1> fausse)
elif ...:                 # succesion de elif
    <Bloc traité>         # si <condition2> vraie (et <condition1> fausse)
else:                     # else ≡ sinon, en anglais
      <Bloc traité>       # si aucune des <conditionN> vraie
<Instructions suivantes>
```


Construire les conditions : opérateurs de comparaison et connecteurs logiques

Déjà présentés en [C0/D14] (plus haut) et [C2/D11], ici plus détaillés

■ **Ce sont des connecteurs pour construire des conditions qui ont le type booléen (`True` / `False`)**

Symbole des opérateurs	Signification en « clair »	Exemple
<	Strictement inférieur (infériorité stricte)	`'a' < 'z'` (True) / `1000 < 999` (False)
>	Strictement supérieur (supériorité stricte)	`'A' > 'B'` (False) / `1000 < 999` (False)
<=	inférieur ou égal (infériorité large)	`2 <= 2` (True) / `10000 <= 1000` (False)
>=	supérieur ou égal (supériorité large)	`2 >= 2` (True) / `10000 >= 1000` (True)
==	Égalité (identité)	`a = 1 ; a == 1` (True) / `'abc' == 'ABC'` (False)
!=	Différent de	`1 != 2` (True) / `a = 2 ; a!=2` renvoient `False`

Connecteurs logiques	Symboles	Exemples
identité logique (oui)	`is`	`'black' is 'white'` renvoie (➛) `False`
complément logique (non)	`not` ou `!`	`not (1!=2)` ➛ `False`
et (and)	`and` ou `&`	`a>b and c>d` ➛ True ou False
ou (or)	`or` ou | (altGr+6 / ⇑+alt+L)	`a >= b or c>d` ➛ True ou False
ou exclusif (xor)	^ (bit à bits)	Pas dans les connexions logiques

Alternative simple « if »

Sur la base d'exemples

S'il n'y a qu'une seule instruction à traiter

```
>>>if p%2 == 0: print('p est pair')
NameError: name 'p' is not defined
>>>p = 4 ; if p%2 == 0: print('p est pair')
p est pair # La réponse est « vraie »
>>>p = 37
>>>if p%2 == 0: print('p est pair')
>>>
```

Les ... indiquent que Python attend d'éventuelles nouvelles instructions. En validant (touche return), nous indiquons à l'interpréteur que le traitement peut commencer.

```
# Pas de réponse : elle est fausse, p impair
```

S'il n'y a plusieurs instructions à traiter, elles forment un bloc qui doit être indenté

```
>>>p = 36
>>>if p%2 == 0 :
...     print('p est pair')
...     File "<stdin>", line 2
...     print('p est pair')
Indentation Error : expected an indented block
>>>if p%2 == 0:
...     print('p est pair')
...     Q = p//2
...     print('p = 2 *', q)
...
p est pair
p = 2 * 18
```

Pas d'indentation...
...provoque une erreur
Mais avec un 'tab', c'est mieux
Le message énoncé en clair

La fin de l'indentation après le print indique que le bloc d'instructions est terminé.

Résultat

Alternative complète « if ... else » et « if ... elif ... else »

Sur un exemple

```
>>>age = 21
>>>if age >= 18 :
...     print('vous êtes majeur')
...else :
...     print('vous êtes mineur')
...
Vous êtes majeur
```

Dans ce cas un message correspondant à une action est affiché : il y aura donc toujours quelque chose d'affiché.
Il y a obligatoirement une indentation avant les instructions (actions) : c'est un bloc.

Et un autre

```
>>>rang=501
>>>if rang <= 50 :
...     print('Vous êtes grand admissible')
...elif rang <= 450 :
...     print('Vous êtes sur liste principale')
...elif rang <= 550 :
...     print('Vous êtes sur liste complémentaire')
...else :
...     print('Vous êtes refusé')
...
Vous êtes sur liste complémentaire
```

Dans ce cas un message correspondant à une action est affiché : comme il y a toujours une option qui est choisie, il y aura toujours quelque chose d'affiché.
Il y a obligatoirement une indentation avant les instructions (actions) : c'est le bloc qui suit les ':'. Chaque bloc peut être composé d'une seule ou de plusieurs actions.

Les structures itératives

(itérations ou boucles)

Les itérations : concept et définitions [C4/D3]

L'idée

- **Dans un programme, il est souvent nécessaire d'exécuter un groupe d'actions (composée ou séquence) quand une variable dont dépendent les actions du groupe évolue**
- **L'« itération » (ou « boucle », ou « répétition ») répond à ce besoin**

Définitions

- **Une boucle est une structure de contrôle qui permet la répétition d'actions un nombre fini de fois qui dépend d'une variable, appelée « indice », qui évolue de manière contrôlée**
- **Le nombre de répétitions peut ne pas être connu *a priori***
  - C'est le cas de l'itération « tant que » ou « `while` » qui utilise une condition (prédicat)
  - Suivant sa valeur de vérité, la condition autorise ou pas les itérations dans la boucle
- **Le nombre de répétitions est clairement connu**
  - C'est le cas de l'itération « pour » ou « `for` » pour laquelle un groupe d'indices est fourni
  - Ce groupe d'indices peut être une plage de valeurs numériques contiguës ou des valeurs prises dans un ensemble (n-uplet ou liste)

Définitions directement issues de l'algorithmique (Cf. diaporama correspondant)

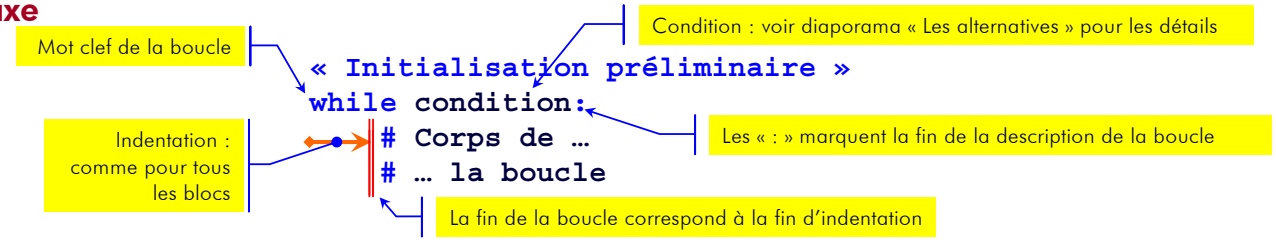
- Les boucles « pour » et « tant que » sont reprises au sein de Python ;
- La boucle « répéter jusqu'à » (`repeat … until`) ne l'est pas.

Structure itérative conditionnelle ou non bornée « `while` » [C4/D5]

Définition, mode de fonctionnement

- La boucle « `while` » permet de répéter un bloc d'instructions tant qu'une condition est vraie.
- Pour « sortir » de la boucle, il faut que la condition finisse par ne plus être vraie.
  - Par conséquent, une variable qui apparaît dans la condition doit changer dans la boucle
- L'évaluation de la condition nécessite de connaître tous ses éléments de la boucle avant son début : il faut donc définir les variables avant. Cette action se nomme « initialiser la boucle ».
- En cas de boucle perpétuelle, c'est une erreur, Python ne s'arrête pas : il faut presser Ctrl & C.

Syntaxe



Exemple : table de multiplication par 7

```
>>> i = 1 # initialiser indice de boucle
>>> while i <= 10:
...     print(i,'* 7 =', i*7,'\t')
...     I += 1
...
1 * 7 = 7   2 * 7 = 14 ... 10 * 7 = 70
```

Initialiser la boucle en fixant l'indice i de départ
Condition vraie, on effectue…
… l'affichage avec print() et …
… l'incrémentation de l'indice de boucle
Les résultats apparaissent…
En les espaçant d'une tabulation à chaque itération.

Quelques informations, exemples et astuces avec les boucles « `while` »

Initialiser est impératif ! Sinon c'est l'erreur assurée

```
>>> while i < 4:
...     print('Et de',i, end='\t\t')
...     i += 1
...
Traceback ... <module>
    while i < 5:
NameError: name 'i' is not defined
```

```
>>> i=0
... while i < 4:
...     print('Et de', i, end='\t\t')
...     i += 1
...
Et de 0   Et de 1   Et de 2   Et de 3
```

Boucle infinie d'attente

- La boucle qui suit ne s'arrête jamais :

```
>>> while True:
...     print('Encore et encore !', end='\t')
Encore et encore !    Encore et encore !    Encore et encore !    Encore et encore ! (etc.)
```

- Ou alors il faut presser Ctrl & C. (ou cliquer le bouton d'arrêt ■)

La variété des conditions

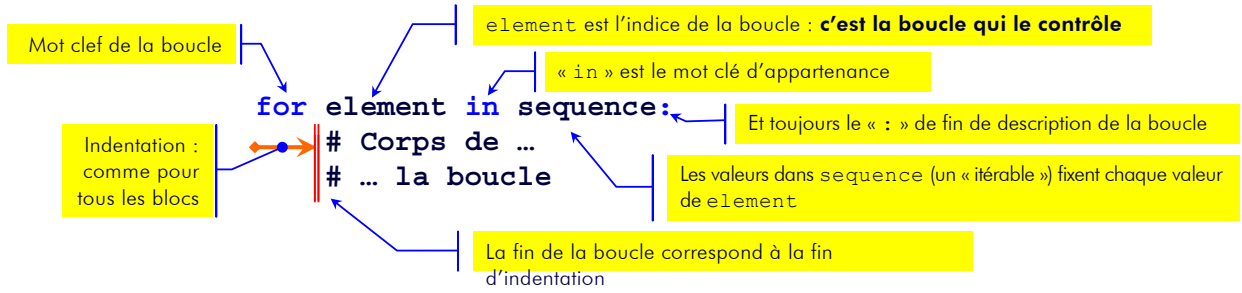
- Tout ce qui a une valeur de vérité, vrai (`True`) ou faux (`False`)
- Des combinaisons diverses sont possibles : voir celles exposées dans « alternatives ».

Structure itérative inconditionnelle ou bornée « `for` » [C4/D9à11]

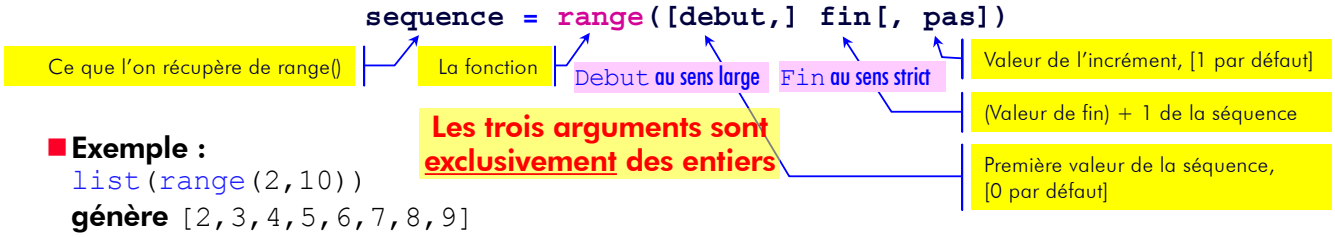
Définition, mode de fonctionnement

- La boucle « `for` » permet de répéter un bloc d'instructions d'après une séquence de valeurs.
- Ces variables de « séquence » ou des « itérables » sont des suites d'entiers, des chaînes, des n-uplets (tuple) ou des liste.
- À la différence de la boucle « `while` » la boucle « `for` » s'arrête toujours (à un moment)

Syntaxe



Une séquence très répandue est fournie par la fonction « `range()` » à trois arguments :



- **Exemple :**
  `list(range(2,10))`
  génère `[2,3,4,5,6,7,8,9]`

Quelques exemples et astuces pour la boucle inconditionnelle « `for` »

Calcul de la somme des entiers de 1 à 100

```
>>>s = 0
>>>for i in range(1,101):
...     S = s+i
...
>>>s
5050
>>>s == 100*101/2
True
```

Et avec une chaîne ?

```
>>>str = "cool l'info"
>>>for c in str:
...     print(c, end='-')
...
c-o-o-l- -l-'-i-n-f-o-
```

Listes définies en extension et en compréhension

```
>>>liste1 = [1,2,3,4,5]
>>>liste2 = [i for i in range(1,6)]
>>>liste1 == liste2
True
```

Voilà, le schéma des bases du langage Python est dressé

Tout n'est pas dit, mais :

- Des problèmes peuvent être étudiés ;
- Des solutions peuvent être mises sur pieds ;
- Des scripts fonctionnels peuvent être rédigés ;
- Une solution d'évaluation automatique pourra donner des résultats.

Les scripts produits ne seront pas nécessairement…

- optimaux, efficaces à 100 %, élégants sur le plan de la syntaxe,
- mais « ça marchera »

Il sera alors temps de voir les détails avec les cinq diaporamas détaillés

Mais pour l'heure des exemples de rédaction et de résolution de problèmes peuvent être abordés en TD et en TP

Alors, en route !