AIDE MEMOIRE PYTHON

VARIABLES ET TYPES DE BASE

Variables et Affectations

Une variable est une étiquette liée à un objet en mémoire. Elle n'a pas de type et ne nécessite pas de déclaration mais doit être affectée à un objet avant toute utilisation. Son nom ne peut contenir que les caractères **azAZ09_** et ne peut pas débuter par un chiffre. On peut détruire une variable avec le mot clef `del`.

```
variable = value                              # affectation simple
v1, v2, v3 = val1, val2, val3               # affectation multiple
del v1                                        # destruction de v1
```

`None` est une constante pour représenter ce qui n'a pas de valeur, pour autant, affecter `None` à une variable revient bien à lui donner une valeur.

Les opérations sur les types sont :

Opérations	Description
`type(objet)`	Renvoie le type d'un objet
`int(val)`	Transforme une valeur en entier
`float(val)`	Transforme une valeur en nombre à virgule flottante
`str(val)`	Transforme une valeur en chaine de caractère

Type Booléen

`True` et `False` sont les valeurs booléennes. Quelque soit son type, une valeur peut être comparée à un booléen. Les valeurs `None`, `False`, `0`, `0.0`, `''`, `()`, `[]` et `{}` sont considérées comme fausses. Toute autre est vraie.

Les opérations de logique par ordre décroissant de priorité sont :

Opérateurs	Description
x `or` y	Si x **ou** y est `True` alors x `or` y est `True`. Si x **et** y sont `False` alors x `or` y est `False`.
x `and` y	Si x **et** y sont `True` alors x `and` y est `True`. Si x **ou** y est `False` alors x `and` y est `False`.
`not` x	Si x est `True` alors `not` x est `False`. Si x est `False` alors `not` x est `True`.

Opérateurs logiques

Opérations de comparaisons. Le type de chaque opérande d'un opérateur logique (de comparaison) peut être quelconque.Le résultat est un booléen. Les opérations sont combinables : `12 <= x < 20`.

Opérateurs	Description
`<`, `<=`, `>`, `>=`	$<, \leq, >, \geq$
`==`, `!=`	Egalité, différence
`is`, `is not`	Identité d'objet, différence d'objet

Types Numériques

Les trois types numériques de Python sont :

Type	Description	Exemples
`int`	Entiers signés	`1` `-26` `2085`
`float`	Nombres à virgules flottantes	`123.4` `-1.234e2`
`complex`	Nombres complexes	`1 + 1j` `1e0 + 1e0j`

Les principaux opérateurs sur les types numériques sont :

Opérateurs		Descriptions
Arithmétique	`x + y, x - y, x * y`	Somme, soustraction, produit
	`x / y`	Division réelle de x par y
	`x // y`	Division entière de x par y (partie entière du quotient)
	`x % y`	Modulo de x par y (reste de la division entière)
	`x ** y, pow(x, y)`	x puissance y
	`abs(x)`	Valeur absolue de x
	`round(x, n)`	Arrondi de x avec n chiffres après la virgule : `round(3.568769, 3)` ⇒ `3.569`
Binaire	`x & y, x | y, x ^ y`	Opérations bit à bit (et,ou, ou exclusif).
	`x << y, x >> y`	Décalage y bits sur x.
	`~x`	Complément bit à bit (non)

Tous les opérateurs (**+**, **-**, *****, **/**, **//**, ******, **%**, **&**, **|**, **^**) peuvent être utilisés dans une affectation raccourcie.

```
a += b  ⇔  a = a + b
a **= b  ⇔  a = a ** b
a |= b  ⇔  a = a | b
```

SEQUENCES

Description générale

Les séquences sont des suites ordonnées d'objets. Les principales sont : **`tuple`**, **`list`**, **`range`** et **`str`**. Seuls les séquences de type **`list`** sont **mutables** : on peut les modifier. Les séquences sont des objets **itérables** (c'est-à-dire qu'il est possible de les parcourir au moyen d'une boucle **`for`**). On peut accéder aux éléments à partir de leurs indices commençant à 0.

Les quatres principales séquences :

Type	Description	Seq vide	Exemple
`tuple`	Séquence non-mutable de types quelconques	`()`	`(1, 2.2, 'A', None)`
`str`	Séquence non-mutable de caractères	`''`,`""`	`'A', "bonjour"`
`list`	Séquence mutable de types quelconques (tableaux)	`[]`	`[1, 2.2, 3, 5]`
`range`	Séquence non-mutable d'entiers		`range(2, 10, 2)`

Opérations Communes aux Séquences

Opérations	Description
`x in s`	**`True`** si **`s`** contient **`x`**
`x not in s`	**`True`** si **`s`** ne contient pas **`x`**
`s + t`	Concaténation de **`s`** et **`t`**. Interdit pour range
`s * n, n * s`	**`n`** copies superficielles de **`s`** concaténées. Interdit pour range
`s[i]`	Élément d'indice **`i`**. Les indices débutent à 0. `s[-1]` est le dernier élément.
`s[i:j]`	Sous-séquence de l'indice **`i`** à **`j`**
`s[i:j:k]`	Sous-séquence de l'indice **`i`** à **`j`** avec un pas de **`k`**
`len(s)`	Longueur de la séquence **`s`**
`min(s), max(s)`	Valeur minimum et maximum de la séquence **`s`**
`sum(s)`	Somme les éléments contenus dans **`s`**.

La notation se sous-séquence permet d'omettre certaines valeurs :

```
s[:b]                  # de l'indice 0 jusqu'à b
s[a:]                  # de l'indice a jusqu'à la fin
s[:]                # toute la séquence (utile pour dupliquer une seq)
s[::p]                 # tous les p termes de la séquence du début à la fin
s[-1:0:-1]          # la séquence renversée
```

TABLEAUX (TYPE LIST)

Les principales opérations sur les tableaux (type **list**) :

Opérations	Description
`s[i] = x`	Remplace la valeur d'indice **i** par **x**
`s[i:j] = t`	Remplace la sous-séquence de **i** à **j** par les valeurs de l'itérable **t**.
`del s[i:j]`	Équivalent à **`s[i:j] = []`**
`s[i:j:k] = t`	Remplace les objets de **`s[i:j:k]`** par ceux de **t** (même taille).
`del s[i:j:k]`	Équivalent à **`s[i:j:k] = []`**
`s.append(x)`	Ajoute l'élément **x** en fin de la séquence **s**
`s.clear()`	Vide la séquence **s**
`s.copy()`	Crée une copie superficielle de la séquence **s**
`s.insert(i,x)`	Insère l'élément **x** à l'indice **i** (décalage de la fin de la séquence **s**)
`s.pop(i)`	Renvoie et supprime l'élément d'indice **i** de la séquence **s**
`s.remove(x)`	Retire le premier élément tel que **`s[i] == x`**
`s.reverse()`	Renverse l'ordre des éléments de la séquence **s**
`s.sort()`	Trie les éléments de **s**. Uniquement disponible pour le type **list**

DICTIONNAIRES

Un **dict** est une table d'associations clef-valeur. Associer une valeur à une clef existante remplace l'ancienne valeur associée. Une clef est un **objet hashable**. Un **dict** est itérable sur ses clefs. On utilise les accolades pour créer un dict vide **{}** ou rempli : **{key: val, …}**.

Quelques opérations sur les dictionnaires :

Opérations	Description
`len(d)`	Nombre d'associations (taille du dictionnaire)
`d[k]` `d.get(k)`	Renvoie la valeur associée à la clé **k**. S'il n'existe pas de clé **k**, **`d[k]`** lève une exception **`KeyError`** alors que **get** renvoie **None**.
`d[k] = v`	Associe **v** à **k**
`k in d`	Renvoie **True** si la clé **k** existe dans **d**, **False** sinon
`k not in d`	Renvoie **False** si la clé **k** existe dans **d**, **True** sinon
`d.clear()`	Vide le dictionnaire **d**
`d.copy()`	Copie superficielle de **d**
`d.keys()`	Renvoie la liste des clés de **d**
`d.values()`	Revoient la liste des valeurs de **d**
`d.items()`	Renvoie la liste des couples clé-valeur de **d**
`d.popitem()`	Supprime et renvoie la dernière association clé/valeur
`d.pop(k)`	Renvoie la valeur associée à la clé et supprime l'association correspondante

CHAINES DE CARACTERES

Description Générale

Il existe plusieurs notations pour définir des chaînes de caractères (**str**). Elles sont toutes équivalentes : `"Guillemets"`, `'Apostrophes'`, `"""Triple guillemets"""` et `'''Triple apostrophes'''`. Les notations triples peuvent contenir des retour à la ligne, les autres doivent utiliser le caractère échappé : **\n**.

Code	Description
\n	Retour à la ligne
\t	Tabulation

Opération sur les chaînes de caractères

Quelques méthodes sur les chaînes de caractères :

Code	Description
`s.capitalize()`	Renvoie une chaîne avec le premier caractère majuscule et les autres en minuscule
`s.lower()`	Renvoie une chaîne identique ou tous les caractères sont en minuscule
`s.upper()`	Renvoie une chaîne identique ou tous les caractères sont en majsucule
`s.replace(new, old)`	Renvoie une chaîne où les sous-chaînes **`old`** sont remplacées par la sous-chaîne **`new`**
`s.split(sep)`	Renvoie une liste des sous-chaînes en utilisant **`sep`** comme séparateur
`s.isalpha()`	Renvoie `True` si tous les caractères sont alphabétiques.
`s.isnumeric()`	Renvoie `True` si tous les caractères sont numériques

BLOCS DE CODE ET STRUCTURES DE CONTROLE

Un bloc de code est un groupement d'au moins une instruction précédées de "**:**" où chaque instruction est indentée. L'instruction `pass`, ne réalisant rien, permet de creer des blocs ne faisant rien. Un bloc peut en inclure d'autres.

Structures de contrôle :

	Description	Syntaxe

Instruction conditionnelle	Le bloc `if` est exécuté si **`condition`** est `True`. Sinon, le premier des blocs optionnels `elif` associé à une condition `True` sera exécuté. Si toute condition est fausse, le bloc optionnel final `else` sera exécuté. Un seul bloc d'instructions sera exécuté.	`if condition:`
		`  …`
		`elif condition:`
		`  …`
		`else :`
		`  …`

Boucle conditionnelle	Le bloc `while` est répété tant que **`condition`** est `True`. Le mot clef `break` entraine une sortie immédiate de la boucle	`while condition:` `  …`
Boucle comptée	Le bloc est exécuté pour chaque valeur **`i`** contenue dans l'objet **`iterable`**. Le mot clef `break` entraine une sortie immédiate de la boucle.	`for i in iterable:` `  …`

FONCTIONS

Définition

Une fonction est un **bloc d'instructions** (corps d'une fonction) destinés à réaliser une opération. Elle peut recevoir des **paramètres** et renvoie une ou plusieurs valeurs. Une fonction est un objet et peut être manipulé comme toute autre valeur.

```
def nom_fonction(pParam, …, nParam=val, …):
    instructions
    …
    return values
```

Une fonction a toujours une valeur de retour, par défaut `None`. C'est la valeur qui sera substituée à l'appel de la fonction. Pour spécifier la valeur de retour on utilise le mot clé `return` suivit d'une ou plusieurs valeurs regroupées dans un **tuple**.

Paramètres

Une fonction peut n'attendre aucun paramètre ou bien plusieurs. Ces paramètres doivent être présents et dans la bon ordre lors de l'appel de la fonction. Certains paramètres (**`nParam`**) peuvent posséder une valeur par défaut. Si ces paramètres ne sont pas présents lors de l'appel ils prendront la valeur par défaut.

Appel

Pour appeler une fonction, on fait suivre son nom de parenthèses contenant la liste des paramètres en respectant l'ordre décrit dans la définition de la fonction. Les parenthèses sont obligatoires même s'il n'y a aucun paramètre.

```
resultats = nom_fonction(pParam, …, nParam=val, …)
```

Fonction lambda : fonction Anonyme

Une fonction lambda est une fonction d'une seule ligne déclarée sans nom, qui peut avoir un nombre quelconque d'arguments, mais elle ne peut avoir qu'une seule expression. Il arrive souvent qu'une fonction lambda soit passée en argument à une autre fonction.

```
lambda x, y, … : expr(x, y, …)
```

MODULES, PACKAGES ET IMPORT

Un **module** est un **fichier `nom.py`** qui regroupe définitions de constantes, fonctions et classes d'objets. Un **package** est un **ensemble de modules** réunis dans un dossier. On importe modules et packages à partir de leur nom avec l'instruction `import`.

Différentes utilisations pour l'instruction `import`.

Instructions	Descriptions
`import mod`	Importe un module. Notation préfixée pour accéder au contenu : `mod.fonction.`
`from mod import f, g`	Importe seulement les éléments indiqués. Utilisation sans préfixer par le nom du module.
`from mod import *`	Un module entier est importé avec *. Utilisation sans préfixer par le nom du module.

Quelques modules utiles :

Module	Fonctions	Description
`math`	`log(x), log(x, y)`	Logarithme (ou en base **`y`**) de **`x`**.
	`sqrt(x)`	Racine carré de **`x`**.
	`hypot(x,y)`	Hypoténuse de **`x`** et **`y`**.
	`floor(x), ceil(x)`	Partie entière inférieure et supérieure de **`x`**.
	`factorial(x)`	Factorielle de **`x`**.
	`sin(x), cos(x), tan(x),…`	Fonctions trigonométriques usuelles, en radian.
	`degrees(x)`	Conversion de **`x`**, de radians vers degrés
	`randian(x)`	Conversion de **`x`**, de degrés vers radians
`random`	`random()`	Génère un **`float`** suivant une distribution uniforme sur **`[0, 1[`**.
	`randint(a, b)`	Génère un **`int`** suivant une distribution uniforme sur **`[a, b]`**.
	`uniform(a, b)`	Génère un **`float`** suivant une distribution uniforme sur **`[a, b[`**.
	`choice(seq)`	Sélectionne un élément parmi une séquence **`seq`** non vide.
	`shuffle(t)`	Mélange les éléments du tableau **`t`**. Ne renvoie rien.
	`sample(pop, x)`	Renvoie une liste de **`x`** éléments pris la séquence **`seq`**.
`io`	`f=open("nom" [,mode])`	Renvoie une **`file`** correspondant au fichier **`nom`** ouvert en lecture. On peut changer le mode : "**`r`**" (lecture), "**`w`**" (écriture), "**`+`**" (lecture/écriture), "**`a`**" (ajout)…
	`f.write(s)`	Écrit la chaîne **`s`** dans le fichier **`f`**.
	`f.read([n])`	Renvoie **`n`** premiers octets contenus de **`f`**
	`f.readline()`	Renvoie une ligne du fichier **`f`**.
	`f.readlines()`	Renvoie une liste des lignes du fichier **`f`**.
	`f.flush()`	Vide le buffer d'écriture.
	`f.close()`	Vide le buffer d'écriture et ferme le fichier **`f`**.

OPERATIONS AVANCEES

Range

L'instruction `range` permet de construire des séquences contenant des valeurs entières. Elle prend jusqu'à trois paramètres :

```
range(fin)                  # Entiers de 0 à fin-1
range(debut, fin)           # Entiers de debut à fin-1
range(debut, fin, pas)      # Entiers de debut à fin-1 séparés de pas
```

Création de séquences par compréhension (générateurs)

Un générateur est une description d'une suite de valeurs qui ne seront générées qu'à leur utilisation, un générateur consomme donc peu de mémoire. On parle de création par **compréhension**.

```
[exp(x) for x in seq]                   # Ex : [x**2 for x in range(1, 5)] => [1, 4, 9, 16]
[exp(x) for x in seq if condition(x)]   # Ex : [x**2 for x in range(1, 8) if x % 2 == 0] => [4, 16]
```

Ces générateurs permettent également de créer des dictionnaires par **compréhension**.

```
{cle(x):exp(x) for x in seq]    # {str(x):x**2 for x in range(1, 4)} => {'1': 1, '2': 4, '3': 9}
```

Pour créer un tuple par compréhension il faut utiliser la fonction de conversion `tuple`.

```
tuple(exp(x) for x in seq)  # tuple(x**2 for x in range(1, 5))  => (1, 2, 3, 4)
```

Entrées/Sorties Standard

La fonction `print(objects)` permet **d'écrire des chaînes de caractères** sur la sortie standard d'un programme : la console.

La fonction `input(prompt)` permet **d'écrire sur l'entrée standard** : le clavier. Cette fonction renvoie les caractères saisis juqu'au premier retour à la ligne. En option elle prend une chaîne de caractère **`prompt`** qui sera affichée avant.

Lecture/Écriture dans des Fichiers

Les fonctions utilisées pour manipuler des fichiers sont contenues dans le package **`io`**. Il s'agit des fonctions **`open`**, **`close`**, **`read`**, **`readline`**, **`readlines`** et **`write`**. Ces fonctions sont accessibles sans importer explicitement le module **`io`**. Dans l'exemple, suivant la fonction close est appelée implicitement à la sortie du contexte.

Un objet fichier est **itérable** pouvant être utiliser pour lire les lignes une à une.

```
with open("/chemin/du/fichier.ext") as f:
    print(f.readline())         # lit la première ligne de f
    for ligne in f:             # lit toutes les lignes suivantes
        print(ligne)
```

Conversions de Type

Python dispose de fonctions pour les conversions entre type. Les fonctions de conversion vers les types numériques sont `int(objet)`, `float(objet)` et `complex(objet)`.

Pour la conversion en chaîne de caractères, deux fonctions sont disponibles : `str(objet)` et `repr(objet)`. La première donne une représentation lisible, la seconde une représentation sans ambiguité. La fonction str est implicitement appelé lors de l'appel de `print(object)` pour convertir `object` en chaîne de caractères.

La fonction `ord(caractere)` renvoie l'entier correspondant au code Unicode (ou code ASCII) associé au caractère . L'opération inverse est `chr(entier)`.