Valeur, type et variable

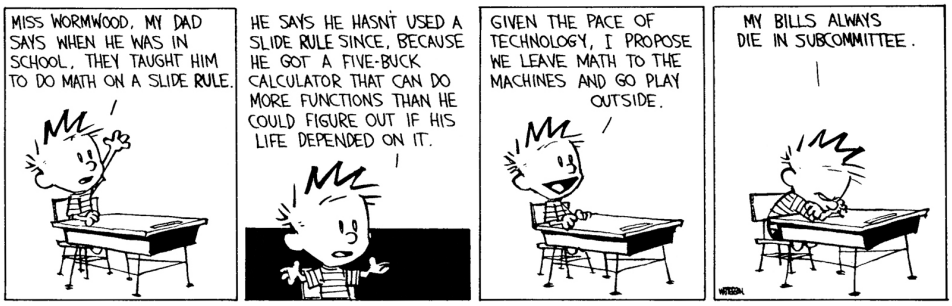

Table des matières

1 Valeur, type

Le langage Python manipule des valeurs de différents *types*. Nous rencontrerons d'abord les types numériques : les *entiers* ainsi que les *nombres flottants* que l'on utilise pour représenter les réels. Nous verrons ensuite les *chaines de caractères*, les *booléens* et les *tuples*.

1.1 Nombre entier

Nous utiliserons Python le plus souvent dans ce qu'on appelle le shell ou la boucle interactive. Ce mode est aussi appelé « Repl » pour : Read, Evaluate, Print, Loop. Autrement dit, lorsqu'on entre une expression, Python la lit, l'évalue, affiche le résultat, et est prêt pour l'interaction suivante.

On écrit les entiers de manière naturelle. D'une manière générale, on obtient le type d'une valeur grâce à la fonction `type`.

```
In [1]: 42
Out[1]: 42

In [2]: type(42)
Out[2]: int
```

Le type des entiers est donc `int`. Les opérateurs usuels d'addition « `+` », de soustraction « `-` », de multiplication « `*` » et d'exponentiation « `**` » sont disponibles pour créer des *expressions* qui sont *évaluées* par l'interpréteur. Ces opérateurs possèdent différents niveaux de priorité. L'exponentiation est prioritaire sur la multiplication. L'addition et la soustraction ont la priorité la plus basse.

```
In [3]: 2 + 3 * 5
Out[3]: 17

In [4]: -1 + 2 ** 8
Out[4]: 255
```

On utilise les parenthèses pour grouper différentes sous-expressions lorsque les calculs que l'on souhaite effectuer diffèrent de ceux fixés par les règles de priorité.

```
In [5]: (2 + 3) * 5
Out[5]: 25
```

Les différentes règles de priorité sont parfois subtiles. Il est donc souhaitable, pour des raisons de lisibilité, d'ajouter des parenthèses dès lors que l'évaluation de notre expression repose sur leur connaissance fine. N'oubliez jamais qu'un programme est écrit pour être lu non seulement par un ordinateur, mais aussi par des humains, qu'ils soient programmeurs ou correcteurs de concours. Par exemple, on n'écrira pas `2 ** 2 ** 3` mais plutôt l'une des deux expressions suivantes :

```
In [6]: (2 ** 2) ** 3
Out[6]: 64

In [7]: 2 ** (2 ** 3)
Out[7]: 256
```

Ces opérateurs sont des opérateurs *binaires* : ils nécessitent deux arguments. Le PEP8, qui fixe les règles de bon usage en Python, recommande de mettre un espace de part et d'autre de tels opérateurs. Cependant, lorsqu'on construit des expressions mélangeant des opérateurs ayant différents niveaux de priorité, il est parfois plus lisible d'omettre cet espace autour des opérateurs ayant la priorité la plus forte. Par exemple, on écrira `2**10 - 1`.

Le symbole « - », utilisé comme opérateur binaire de soustraction, est aussi utilisé pour la négation. Dans ce cas, c'est un opérateur *unaire* ne nécessitant qu'une opérande.

```
In [8]: -2 * 3
Out[8]: -6
```

Contrairement à ce qui se passe dans la plupart des autres langages comme le C ou OCaml, Python peut représenter des nombres aussi grands que l'on souhaite. Prenons l'exemple du n-ième nombre de Mersenne définit par $M_n := 2^n - 1$. Il est courant de chercher des nombres premiers parmi ces entiers. Le dixième nombre de Mersenne qui est premier est M_{89} et son calcul ne pose aucun problème à Python.

```
In [9]: 2**89 - 1
Out[9]: 618970019642690137449562111
```

Python offre deux types de division. Commençons par la division entière. Rappelons le théorème de la division euclienne sur $\mathbb{Z}$:

Proposition 1.1

Soit $a \in \mathbb{Z}$ et $b \in \mathbb{N}^*$. Alors il existe un unique couple $(q, r) \in \mathbb{Z}^2$ tel que

$$a = qb + r \quad \text{et} \quad 0 \leqslant r < b.$$

q est appelé *quotient* de la division euclidienne de a par b, et r son *reste*.

Par exemple $7 = 2 \times 3 + 1$, donc 2 est le quotient de la division euclidienne de 7 par 3 et son reste est 1. De même $-7 = (-3) \times 3 + 2$, donc -3 est le quotient de la division euclidienne de -7 par 3 et son reste est 2. En Python, on obtient le quotient de la division euclidienne de a par b avec `a // b` et son reste avec `a % b`.

```
In [10]: -7 // 3
Out[10]: -3

In [11]: -7 % 3
Out[11]: 2
```

La division par 0 est une erreur et lève ce qu'on appelle une *exception*. Même s'il est possible de rattraper les exceptions, nous ne le ferons pas dans ce cours et une division par 0 aura pour effet de produire l'erreur suivante :

```
In [12]: 1 // 0
ZeroDivisionError: integer division or modulo by zero
```

Python offre aussi une division plus classique, notée `/`. Elle produit une valeur d'un type différent : celui des nombres flottants.

```
In [13]: 3 / 2
Out[13]: 1.5

In [14]: type(1.5)
Out[14]: float
```

1.2 Nombre flottant

Les nombres flottants sont utilisés pour représenter les nombres réels. Comme tous les langages de programmation, Python utilise le « . » comme séparateur décimal. Pour calculer une valeur approchée de la circonférence d'un cercle de diamètre 2, on entre donc :

```
In [1]: 3.14 * 2.0
Out[1]: 6.28
```

Commençons par remarquer que les opérateurs `+`, `-`, `*`, `/` et `**` sont disponibles pour les nombres flottants. On peut par ailleurs mélanger flottants et entiers dans les calculs. Comme pour les entiers, la division par `0.0` lève une exception.

Pour simplifier l'écriture de grands et de petits nombres, on utilise la notation scientifique. Ainsi, l'âge de l'univers étant estimé à 13.8 milliards d'années et la vitesse de la lumière étant de l'ordre de 3.0×10^8 mètres par seconde, le calcul suivant nous montre qu'il est impossible d'observer des endroits de l'univers à une distance supérieure à 1.31×10^{26} mètres de la terre :

```
In [2]: 13.8e9 * 365 * 24 * 60 * 60 * 3e8
Out[2]: 1.3055904e+26
```

Pour le calcul de la circonférence du cercle de diamètre 2, on a approché plus haut π par 3.14. On pourrait bien sûr utiliser une approximation plus précise comme 3.14159,

```
In [3]: 2 * 3.14159
Out[3]: 6.28318
```

mais la précision disponible avec Python n'est pas illimitée. En première approximation, on peut considérer que Python ne peut travailler qu'avec des nombres flottants ayant une précision de 16 chiffres significatifs : tout excès de précision est ignoré.

```
In [4]: 1234567890.12345678 - 1234567890.1234567
Out[4]: 0.0
```

Le premier nombre possède 18 chiffres significatifs alors que le second en possède 17. Avant même d'effectuer la soustraction, les deux nombres sont arrondis au même nombre : le résultat final est donc nul. La situation est en fait plus complexe que cela, car tout comme les entiers, les flottants ne sont pas représentés en interne en base 10 mais en base 2. Ne soyez donc pas surpris si, en faisant vos propres essais, vous avez parfois l'impression que Python garde 16 chiffres significatifs, parfois 17.

Pour les mêmes raisons, le résultat de chaque opération arithmétique est arrondi. Cela conduit à des résultats surprenants comme le calcul suivant qui n'est pas égal à 10^{-16} comme on pourrait s'y attendre.

```
In [5]: (1.0 + 1.0e-16) - 1.0
Out[5]: 0.0
```

En effet, $1.0 + 10^{-16}$ possède 17 chiffres significatifs. Il est arrondi à 1.0 avant d'effectuer la soustraction qui donne donc 0. Comme les flottants sont stockés en base 2 et non en base 10, même les nombres décimaux les plus simples ne sont pas représentables exactement. On peut donc avoir des résultats surprenants :

```
In [6]: 0.1 + 0.2 - 0.3
Out[6]: 5.551115123125783e-17
```

Vous comprendrez pourquoi les logiciels de comptabilité ne travaillent pas en interne avec des nombres flottants. Ils ont cependant de nombreuses qualités et sont utilisés en simulation numérique ainsi qu'en intelligence artificielle. On n'oubliera cependant jamais que des arrondis sont effectués à chaque opération et nous verrons que les erreurs accumulées peuvent parfois devenir significatives et fausser complètement un résultat.

Lorsqu'on mélange des entiers et des flottants dans une expression, une conversion préalable des entiers vers les flottants est réalisée automatiquement. Cette conversion automatique est un choix raisonnable, car contrairement aux nombres décimaux, les nombres entiers qui ne sont pas trop grands (disons, ceux qui s'écrivent avec moins de 16

chiffres) sont représentables de manière exacte par des flottants. La conversion des flottants vers les entiers est aussi possible. Cependant, comme elle fait perdre de l'information, il faut la demander explicitement en utilisant la fonction `int`. Cette fonction arrondit un flottant au premier entier rencontré lorsqu'on se rapproche de 0.

```
In [7]: int(2.718)
Out[7]: 2

In [8]: int(-2.718)
Out[8]: -2
```

De manière générale, chaque type numérique possède une fonction associée pour forcer une conversion.

Les fonctions usuelles sont disponibles dans la bibliothèque `math`. On y trouve par exemple la fonction `floor` qui, pour chaque nombre x, renvoie sa partie entière dont on rappelle la définition ci-dessous.

Proposition 1.2

Soit $x \in \mathbb{R}$. Il existe un unique $n \in \mathbb{Z}$ tel que

$$n \leqslant x < n+1.$$

Cet entier est appelé *partie entière* de x et est noté $\lfloor x \rfloor$.

Pour charger la bibliothèque `math`, on utilise l'instruction suivante :

```
In [9]: import math

In [10]: math.floor(2.718)
Out[10]: 2

In [11]: math.floor(-2.718)
Out[11]: -3
```

De même, on définit la partie entière supérieure d'un réel.

Proposition 1.3

Soit $x \in \mathbb{R}$. Il existe un unique $n \in \mathbb{Z}$ tel que

$$n-1 < x \leqslant n.$$

Cet entier est appelé *partie entière supérieure* de x et est noté $\lceil x \rceil$.

La fonction `ceil` de la même bibliothèque permet d'y accéder.

```
In [12]: math.ceil(2.718)
Out[12]: 3

In [13]: math.ceil(-2.718)
Out[13]: -2
```

On peut calculer la racine carrée d'un nombre avec la fonction `sqrt`, abréviation de « square root ». Les autres fonctions usuelles sont aussi disponibles.

```
In [14]: math.sqrt(2.0)
Out[14]: 1.4142135623730951

In [15]: math.exp(1.0)
Out[15]: 2.718281828459045
```

Le logarithme naturel ln (`log` en Python), le logarithme en base 10 (`log10`) et le logarithme en base 2 (`log2`) sont aussi disponibles. Bien sûr, les fonctions trigonométriques circulaires cos, sin et tan sont présentes, tout comme la constante π.

```
In [16]: math.cos(math.pi / 17)
Out[16]: 0.9829730996839018
```

La principale devise de Python est « batteries included ». Autrement dit, de nombreuses bibliothèques (« libraries » en anglais), sont disponibles. Nous venons d'utiliser notre première bibliothèque : le module `math`. Il existe de nombreuses manières de les rendre accessibles. La plus simple est d'écrire `import` suivi du nom de la bibliothèque. Les fonctions et constantes seront alors disponibles, préfixées par le nom du module. Si vous souhaitez seulement en utiliser certaines sans avoir à taper à chaque fois le nom du module, vous pouvez entrer la commande :

```
In [17]: from math import cos, pi

In [18]: cos(pi / 17)
Out[18]: 0.9829730996839018
```

Il est d'ailleurs possible d'importer tous les composants du module `math` avec la commande

```
In [19]: from math import *
```

C'est cependant une opération dangereuse, car si on importe ainsi plusieurs modules, on ne sait rapidement plus d'où viennent nos fonctions. En pratique, il est préférable d'utiliser `import math`, quitte à renommer le module en un nom plus court. On utilise pour cela la commande :

```
In [20]: import math as ma

In [21]: ma.cos(ma.pi / 3)
Out[21]: 0.5000000000000001
```

1.3 Chaine de caractères

Python nous permet de travailler avec du texte. Pour cela, on utilise des chaines de caractères que l'on encadre en utilisant soit des « " », soit des « ' ».

```
In [1]: "hello, world"
Out[1]: 'Hello, world'

In [2]: 'Ça dépend, ça dépasse.'
Out[2]: 'Ça dépend, ça dépasse.'
```

Python permet d'utiliser des accents dans les chaines de caractères. Vous pouvez même utiliser des caractères espagnols, allemands, russes, hébreux, arabes ou chinois. Bref, tous les caractères UNICODE sont supportés. De nombreux caractères « spéciaux » existent. Par exemple, le retour à la ligne est un « caractère » que l'on obtient en écrivant « `\n` ». De même, la tabulation est un caractère que l'on obtient en écrivant « `\t` ». La plupart des éditeurs de texte affichent la tabulation en la remplaçant par 2 ou 4 espaces, mais il est important d'être conscient que dans les fichiers textes, c'est un caractère à part entière. Comme il est difficile de le distinguer d'une succession d'espaces, on convient en général de ne pas l'utiliser : beaucoup d'éditeurs de texte insèrent des espaces plutôt qu'un caractère de tabulation lorsque l'on utilise la touche « TAB » de notre clavier. Enfin, si vous souhaitez utiliser des guillemets dans une chaine de caractères et que votre choix du délimiteur vous en empêche, vous pouvez l'insérer avec « `\"` » ou « `\'` ». Par exemple :

```
In [3]: "What do you mean \"ew\"? I don't like Spam!"
Out[3]: 'What do you mean "ew"? I don't like Spam!'
```

Il est possible d'obtenir la longueur d'une chaine de caractères en utilisant la fonction `len`.

```
In [4]: len("hello, world")
Out[4]: 12
```

Les chaines de caractères ont leur propre type : le type `str`.

```
In [5]: type("Il suffit pas d'y dire, y faut aussi y faire.")
Out[5]: str
```

On ne confondra pas les chaines de caractères et les entiers. En particulier, si on essaie d'ajouter un entier à une chaine de caractères en écrivant « `2 + "2"` », on obtient une erreur de type. Cependant, on peut utiliser le symbole `+` entre deux chaines de caractères, ce qui a pour effet de les concaténer :

```
In [6]: "Tic" + "Tac"
Out[6]: 'TicTac'
```

De la même manière, il est possible de multiplier une chaine de caractères par un entier.

```
In [7]: "G" + 10 * "o" + "al"
Out[7]: 'Gooooooooooal'
```

On peut convertir une chaine de caractères en un entier ou un nombre flottant. C'est une conversion de type et il suffit pour cela d'appliquer la fonction portant le nom du type désiré à notre chaine.

```
In [8]: int("2")
Out[8]: 2

In [9]: float("13.1")
Out[9]: 13.1

In [10]: int("2") * float("13.1")
Out[10]: 26.2
```

Si la chaine de caractères ne peut être interprétée comme une valeur du type demandée, une exception sera levée. La conversion inverse est possible avec la fonction `str` :

```
In [11]: "Fahrenheit " + str(45) + str(1)
Out[11]: 'Fahrenheit 451'
```

Le standard Ascii associe une valeur entre 0 et 127 à chacun des caractères les plus courants. Le tableau ci-dessous donne ces valeurs, les cases grisées représentant des caractères non imprimables. Par exemple, le caractère A est associé à la valeur 65.

	0	1	2	3	4	5	6	7	8	9
0										
10										
20										
30				!	"	#	$	%	&	’
40	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	\	]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{	|	}	~			

Pour obtenir cet entier, on utilise la fonction `ord`.

```
In [12]: ord('A')
Out[12]: 65
```

On peut obtenir un caractère à partir de son code Ascii grâce à la fonction `chr`.

```
In [13]: chr(65)
Out[13]: 'A'
```

1.4 Booléen

Python possède un type booléen qui n'a que deux valeurs distinctes : `True` et `False`.

```
In [1]: True
Out[1]: True

In [2]: type(True)
Out[2]: bool
```

Les opérateurs logiques usuels « et », « ou » et « non » sont disponibles. On remarquera que le « ou » est bien le ou inclusif, comme en mathématiques.

```
In [3]: True and True
Out[3]: True
```

```
In [4]: True and False
Out[4]: False

In [5]: True or True
Out[5]: True

In [6]: not True
Out[6]: False
```

Pour savoir si deux valeurs sont égales, on utilise le symbole « == ».

```
In [7]: 1 + 1 == 2
Out[7]: True
```

La valeur renvoyée par un test d'égalité est un *booléen*. Attention à ne jamais utiliser de test d'égalité entre deux flottants, car les arrondis auxquels ils sont sujets font que certains résultats sont surprenants !

```
In [8]: 0.1 + 0.2 == 0.3
Out[8]: False
```

En général, deux valeurs de types différents ne sont pas égales.

```
In [9]: 2 == "2"
Out[9]: False
```

Les opérateurs `!=`, `<`, `>`, `<=`, `>=` sont aussi disponibles.

`x == y`	x est égal à y
`x != y`	x est différent de y
`x < y`	x est strictement inférieur à y
`x > y`	x est strictement supérieur à y
`x <= y`	x est inférieur ou égal à y
`x >= y`	x est supérieur ou égal à y

```
In [10]: 3 <= 3.14 and 3.14 <= 4
Out[10]: True
```

Pour la comparaison des chaines de caractères, l'ordre utilisé est l'ordre lexicographique, chaque caractère étant ordonné dans l'ordre de la table Ascii/Unicode.

```
In [11]: "OL" > "OM"
Out[11]: False
```

Faites attention à l'ordre de ces caractères. Les minuscules sont bien évidemment dans l'ordre alphabétique, tout comme les majuscules, mais la lettre « Z » est avant la lettre « a » et donc de manière surprenante `"Zorro" < "algèbre"`.

Exercice 1

⇨ En utilisant le tableau Ascii, classer ces chaines de caractère dans l'ordre lexicographique : `"9"`, `"34"`, `"Maison"`, `"la"` et `"laisser"`.

1.5 Tuple

Afin de grouper plusieurs valeurs, Python propose un type appelé `tuple`.

```
In [1]: (1.0, 2.0)
Out[1]: (1.0, 2.0)

In [2]: "Teddy", "Riner", 1989
Out[2]: ("Teddy", "Riner", 1989)

In [3]: type((2.0, 1.0))
Out[3]: tuple
```

Les parenthèses regroupant ces valeurs sont optionnelles. Il est courant d'utiliser des tuples pour grouper des valeurs n'ayant pas le même type, comme dans notre second exemple.

2 Programmation impérative

2.1 Variable

Les valeurs déjà calculées peuvent être gardées en mémoire afin de les utiliser plus tard. Pour cela, on utilise des variables. Les types que nous avons vus jusqu'ici seront plus tard décrits comme *immuables* et lorsqu'on travaille avec de tels types, une variable peut être conceptualisée par une boite portant un *nom* et contenant une *valeur*. Afin de stocker une valeur dans une boite, on utilise le symbole d'*affectation* « `=` ».

```
In [1]: a = 6
```

À gauche du symbole d'affectation, on place le nom de la boite qui doit être utilisée. À droite, on doit trouver une expression qui sera évaluée en une valeur. Cette valeur sera alors stockée dans la boite.

On accède ensuite à la valeur mémorisée en utilisant le nom de la variable. Lors de l'évaluation de chaque expression, les noms de variables sont remplacés par les valeurs qu'elles contiennent.

```
In [2]: a * (a + 1)
Out[2]: 42
```

Exercice 2

⇨ Dans cet exercice on s'interdit d'utiliser l'exponentiation « `**` ».

1. Montrer que l'on peut calculer a^8 avec 3 multiplications.
2. Donner une manière de calculer a^7 avec 4 multiplications.

Une fois qu'une variable est définie, il est possible de la redéfinir en utilisant une nouvelle valeur.

```
In [3]: a = 7

In [4]: a = a + 1

In [5]: a
Out[5]: 8
```

Pour l'entrée `a = a + 1`, le membre de droite est d'abord évalué pour produire la valeur 8. Cette valeur est ensuite stockée dans la variable `a`. L'ancienne valeur est « écrasée » et il n'est plus possible d'y accéder. Ce type d'instruction nous rappelle que le symbole d'affectation est dissymétrique, contrairement au symbole d'égalité utilisé en mathématiques. En particulier, l'instruction « `a + 1 = a` » n'a aucun sens et sera signalée par Python comme une erreur. Remarquons que c'est bien une valeur qui est stockée dans une variable. En particulier, si l'on définit « `b = a` » et que l'on change ensuite la valeur de `a`, celle de `b` reste inchangée.

Exercice 3

⇨ La méthode de Héron est une méthode historique pour obtenir une valeur approchée de la racine carrée d'un nombre $a > 0$. Pour cela, on définit la suite (u_n) par

$$u_0 \coloneqq a, \qquad \text{et} \qquad \forall n \in \mathbb{N}, \quad u_{n+1} \coloneqq \frac{u_n + \frac{a}{u_n}}{2}.$$

Déterminer la plus petite valeur de n pour laquelle la précision sur les nombres flottants ne permet plus de distinguer u_n de u_{n+1} lors du calcul de $\sqrt{2}$.

Python est un langage de programmation à *typage dynamique* : une même variable peut à un moment donné stocker un entier et plus tard une chaine de caractères. Le type d'une variable, c'est-à-dire le type de la valeur stockée par cette variable est donc autorisé à changer lors de l'exécution d'un programme. Cette manière de programmer rend cependant les programmes plus difficiles à lire et nous éviterons de le faire.

Pour les noms de variables, nous nous limiterons aux noms composés de lettres minuscules (`a-z`) et majuscules (`A-Z`), ainsi qu'au caractère « tiret du bas » ou « underscore » (`_`) disponible sur la touche 8 des claviers français. L'utilisation de chiffres (`0-9`) à la fin d'un nom est autorisée. On évitera d'utiliser les accents dans les noms de variables. Choisir judicieusement le nom de ses variables est un art qu'il est important de cultiver. Les noms de variables courts ont l'avantage d'être rapides à taper et à lire. On les utilisera donc pour stocker des valeurs que nous utiliserons souvent. Les noms de variables longs ont l'avantage d'être plus descriptifs. On les utilisera donc pour faire référence à des valeurs que nous utiliserons plus rarement. Pour des noms de variables composés de plusieurs mots, on utilise souvent un underscore comme dans `nb_eleves` ou une lettre majuscule comme dans `nbEleves`.

2.2 État du système

Contrairement aux expressions dont la finalité est de produire une valeur, une affectation a pour effet de changer l'état des variables. On dit qu'elle agit par *effet de bord*. Pour représenter l'état du système, nous utiliserons la notation suivante `{a: 2, b: 7}`. Elle signale que la variable `a` contient la valeur 2 tandis que `b` contient la valeur 7. La programmation *impérative* consiste à écrire une succession d'instructions pour changer l'état de la machine. Le langage machine, qui est utilisé par les processeurs, fonctionne de cette manière. C'est une des raisons pour lesquelles ce style est central dans de nombreux langages de programmation. C'est le cas pour Python, et c'est un style que nous adopterons souvent dans ce cours. Afin de visualiser l'état dans lequel se trouve la machine, on le décrira sur une ligne de commentaire. Ces lignes commencent par le caractère `#` et sont ignorées par Python.

Si par exemple les variables `a` et `b` contiennent respectivement 2 et 7, les instructions suivantes modifient l'état de la machine comme suit :

```
# etat {a: 2, b: 7}
In [1]: a = b
In [2]: b = a
# etat {a: 7, b: 7}
```

En particulier, ces deux instructions n'ont pas eu pour effet d'échanger le contenu des variables `a` et `b`. La première instruction a eu pour effet d'écraser la valeur contenue dans `a` qui est alors définitivement perdue. Si on possède un verre d'eau et un verre de vin, le meilleur moyen pour échanger le contenu de ces verres est d'utiliser un troisième verre. Pour échanger deux variables, on peut donc utiliser la séquence d'instructions suivante :

```
# etat {a: 2, b: 7}
In [1]: c = a
In [2]: a = b
In [3]: b = c
# etat {a: 7, b: 2, c: 2}
```

Notons que Python permet d'utiliser les `tuple` pour effectuer des affectations « simultanées ». Par exemple après l'instruction

```
In [4]: a, b = 7, 2
```

`a` contient 7 et `b` contient 2. Puisque l'expression de droite est évaluée avant l'affectation, cette construction est très utile pour échanger le contenu de deux variables.

```
In [5]: a, b
Out[5]: (7, 2)

In [6]: a, b = b, a

In [7]: a, b
Out[7]: (2, 7)
```

En pratique, on réservera ces affectations simultanées aux cas où plusieurs affectations les unes à la suite des autres ne permettent pas d'obtenir un résultat similaire.

Notons qu'il est possible de supprimer une variable avec l'instruction `del`, mais nous nous contenterons d'ignorer les variables dont nous n'avons plus l'utilité.

On notera parfois $\mathcal{E}_0, \mathcal{E}_1, \ldots$ l'état du système à différentes étapes de l'exécution de notre code. On utilisera aussi la convention suivante : si `a` est une variable, a_k sera la valeur contenue par cette dernière quand le système est dans l'état $\mathcal{E}_k$. Par exemple

```
# etat0 {a: a0, b: b0}
In [8]: a = a + b
# etat1 {a: a0 + b0, b: b0}
```

signifie qu'après notre affectation $a_1 = a_0 + b_0$ et $b_1 = b_0$.

2.3 Entrée, sortie

Le langage Python permet d'interagir avec l'utilisateur en demandant d'entrer des valeurs avec lesquelles il va travailler puis en affichant les résultats de son calcul.

Pour afficher une valeur, on utilise la fonction `print`. On l'utilise pour afficher des chaines de caractères, mais aussi des entiers ou des nombres flottants.

```
In [1]: print("hello, world")
hello, world

In [2]: print(2**10)
1024
```

La fonction `print` travaille par effet de bord. Elle a pour effet de changer l'état du système, à savoir ce qui est affiché par la *console* de l'ordinateur. Il est essentiel de bien faire la différence entre l'expression `2**10` qui s'évalue en 1024 et l'appel `print(2**10)` qui affiche 1024 sur la console. Cet appel renvoie `None`, l'unique valeur du type `NoneType` qui est renvoyée par les fonctions travaillant par effet de bord. On ne voit pas cette valeur sur une ligne `Out` car le shell a pour habitude de ne jamais l'afficher. La différence peut paraitre subtile lorsque l'on travaille avec Python en mode interactif mais elle existe bien :

```
In [3]: 2**10
Out[3]: 1024

In [4]: print(2**10)
1024
```

La fonction `print` peut être utilisée avec plusieurs valeurs, séparées par des virgules : elles sont affichées sur une même ligne, les unes à la suite des autres.

```
In [5]: nb_eleves = 43

In [6]: print("Il y a", nb_eleves, "éleves dans la classe.")
Il y a 43 élèves dans la classe.
```

Par défaut, un retour à la ligne est automatiquement ajouté après chaque appel à `print`. Pour éviter cela, on peut utiliser l'option `end` et remplacer le caractère « `\n` », par la chaine de votre choix. Le plus courant est d'utiliser une chaine vide.

```
In [7]: print("hello, ", end="")
   ...: print("world")
hello, world
```

Enfin, lorsque vous voulez afficher plusieurs valeurs sur une même ligne en les séparant par des virgules, Python va ajouter un espace entre chaque valeur. Cela peut être utile, mais dans les cas où vous ne le souhaitez pas, vous pouvez construire les chaines de caractères à la main.

```
In [8]: n = 3

In [9]: print("Sup" + str(n) + " rocks!")
Sup3 rocks!
```

La fonction `input` permet quant à elle de demander des valeurs à l'utilisateur. Quelle que soit la valeur attendue, c'est sous la forme d'une chaine de caractères que Python la renvoie au programmeur. Il convient donc d'effectuer explicitement une conversion lorsque l'on souhaite une valeur d'un autre type.

```
In [10]: nom = input("Quel est votre nom ? ")
Quel est votre nom ? Teddy Riner

In [11]: entree = input(nom + ", quelle est votre année de naissance ? ")
Teddy Riner, quelle est votre année de naissance ? 1989

In [12]: annee = int(entree)
In [13]: print("Vous aurez", 2024 - annee, "ans l'année des jeux de Paris.")
Vous aurez 35 ans l'année des jeux de Paris.
```

Bien qu'elle renvoie une valeur, on dit aussi que la fonction `input` travaille par effet de bord, car elle attend une entrée de l'utilisateur. C'est la seule fois dans ce cours où vous verrez cette fonction. Nous sommes en 2023, les téléphones ont des interfaces tactiles et la reconnaissance vocale est plutôt efficace. Tout cela pour vous dire qu'il vaut mieux laisser gérer l'interface utilisateur par des personnes maitrisant ces technologies. Comme nous travaillerons en mode interactif, `print` et `input` nous seront de toute façon le plus souvent inutiles et nous vous demandons de les laisser de côté, sauf si on vous demande explicitement de les utiliser.