Feuille de TP Informatique 02

Calculs en bases ; Expressions, variables et types

Exercice 1.—
1. Donner l'écriture décimale de $\underline{10\,1111}$. On pourra vérifier le résultat avec Python (en saisissant `0b101111` dans le shell).
2. Donner l'écriture en binaire naturel de 55. On pourra vérifier avec Python (saisir `bin(55)` dans le shell).
3. Donner l'écriture décimale de (écrit en base 8) 7413.
4. Donner l'écriture en base 3 de 63.

Exercice 2.—
1. Comment voir simplement si un entier écrit en binaire naturel est pair ou impair ?
2. Combien de chiffres 0 ou 1 sont nécessaires pour écrire en binaire naturel tous les chiffres d'une heure de la journée, au format heures-minutes-secondes, avec un nombre d'heures compris entre 0 et 23 ?
3. Quelle est la plus grande factorielle (nombres de la forme $n! = 1 \times 2 \times \ldots \times (n-1) \times n$) que l'on peut écrire en binaire naturel avec 8 chiffres ? Avec 16 chiffres ?

Exercice 3.—
1. Donner l'expression en binaire naturel de $\underline{10\,1100}$+$\underline{1\,1001}$ (sans passer par les décimaux).
2. Écrire la table en binaire naturel de tous les entiers compris entre 0 et 16 inclus (sans les calculer un par un, mais en ajoutant 1 à chaque fois).

Exercice 4.—
1. À quelle opération est équivalent le fait de décaler tous les bits d'un entier non signé vers la gauche, en rajoutant un 0 à droite (exemple : $\underline{10100} \to \underline{101000}$ ou encore $\underline{1001} \to \underline{10010}$) ?
2. À quelle opération est équivalent le fait de décaler tous les bits d'un entier non signé vers la droite, en supprimant le bit de poids faible (exemple : $\underline{10100} \to \underline{1010}$ ou encore $\underline{1001} \to \underline{100}$) ?

Exercice 5.—
1. Quelle est l'écriture en décimal du nombre *D2F* en écriture hexadécimale ? On pourra vérifier le résultat avec Python (`0xD2F`).
2. Écrire sous forme hexadécimale le nombre 244. Vérifier avec Python (`hex(244)`).
3. Une clef Wifi s'écrit *D336 45E4 4FA6 43BC 44F9 FA79 A6* en hexadécimal. Combien de chiffres 0 et 1 faudrait-il pour l'écrire en binaire naturel ? Combien de chiffres nécessite l'écriture décimale de ce nombre (ne pas faire le calcul à la main : utiliser Python !) ? En déduire les avantages de la numération hexadécimale dans ce contexte.

Exercice 6.— Évaluer en Python les expressions suivantes –on pourra, au besoin, préalablement créer une ou plusieurs variables flottantes– en utilisant les fonctions mathématiques du module `numpy` permettant leur évaluation :
1. $\ln(10)$;
2. $e^{-1/(1-x^2)}$ avec $x = 0,5$;
3. $\binom{100}{50} = 100!/(50!)^2$ où $n! := n \times (n-1) \times \ldots \times 2 \times 1$ désigne la factorielle de n (Vérifier que ce nombre est essentiellement le même que $\dfrac{2^{100}}{\sqrt{50.\pi}}$.) ;
4. $k = 2\pi/\lambda$ avec λ = 20 cm.

Exercice 7.—
On considère l'extrait de script Python suivant, où les variables `a` et `b` sont de type numérique (des flottants par exemple) :

```
a = 4.125 # ou n'importe quel autre flottant
b = -3.88 # ou n'importe quel autre flottant
a = a+b
b = a-b
a = a-b
```

1. Par rapport à leurs valeurs initiales, quelles sont les valeurs de `a` et `b` après l'exécution du script ?
2. Vérifier la réponse à la question précédente en recopiant le script et en l'exécutant (rappel : dans IEP ou Spyder, l'exécution en bloc d'un script se fait à l'aide de la touche F5).
3. Pourquoi parle-t-on ici de calcul « sans mémoire additionelle » ?
Comment aboutit-on au même résultat en utilisant un peu de mémoire additionnelle ?
4. En Python, comment peut-on, en une ligne, aboutir au même résultat ?

Exercice 8.— Donner le type et la valeur des expressions suivantes (de tête puis vérification en console) :
1. `44 >= 91//2`
2. `[(i**2+11)%7 for i in range(1,10)]`
3. `6400/3200`
4. `'Ce n\'est pas la peine'+" de discuter là-dessus maintenant, Apollodore."`
5. `('Ce n\'est pas la peine'+" de discuter là-dessus maintenant, Apollodore.")[3]`
6. `[8,9,10][2]`
7. `int(5.6)`
8. `float(5)`
9. `len(str(455.8))`

Exercice 9.— Si `a`, `b` et `c` sont des variables entières, l'expression `a**b**c` est-elle égale à `(a**b)**c` ou à `a**(b**c)` ? Pour répondre, on pourra faire des tests dans le shell pour des valeurs arbitraires de `a`, `b` et `c`.

Qu'en est-il si l'on remplace `**` par `//` ?

Exercice 10.—
1. Prévoir les valeurs des expressions et vérifier sur l'ordinateur :
1.a. `False or False and not True`
1.b. `True or not True and False`
1.c. `17*19 != 324 or not True == False`
2. L'expression `1000 < 10 < 100` a-t-elle du sens en Python ? Même question pour `1000 > 10 < 100` ainsi que pour `1000 < 10 < 1/0`.
3. Selon Python, quelle est la plus grande des constantes `True` et `False` Expliquer comment vous avez obtenu votre réponse.
4. Soit `mot` une variable de type chaîne de caractères. Proposer une expression dont la valeur est `True` si `mot` contient toutes les voyelles de l'alphabet (`a`, `e`, `i`, `o`, `u`, `y`), et `False` sinon. Faire un test avec `mot` égal à

```
"Portez ce vieux whisky au juge blond qui fume."
```

et avec

```
"A man, a plan, a canal : Panama."
```

(qui au passage, sont respectivement un célèbre pangramme et un palindrome).

Exercice 11.—
1. On considère la chaîne de caractères `"Le neutron est aussi une onde."`, affectée à une variable dont l'identifiant est `chaine`.

Prévoir les valeurs des expressions suivantes, et vérifier ces prévisions à l'aide de Python :
1.a. `len(chaine)` **1.b.** `chaine[5]` **1.c.** `chaine[3:16]` **1.d.** `chaine[3:16:4]`
1.e. `chaine[-1]` **1.f.** `chaine[-2]` **1.g.** `chaine[-3]` **1.h.** `chaine[10:]`
1.i. `chaine[:10]` **1.j.** `chaine[:]` **1.k.** `chaine[::-1]`
2. Quelle est la valeur de l'expression `len("C\'est trop un truc de ouf.")` ?
3. Comment déclarer une chaîne contenant un `\` ?
4. Quelle est la différence entre les chaînes `"Je saute à la ligne"` et `"Je saute à la ligne\n"` ?

Exercice 12.—
1. On considère la liste `liste` égale à `[1,2,3,[4,5,6,7,8],[9,10,11,12],13,14,15,16]`. Prévoir les valeurs des expressions suivantes, et vérifier ces prévisions à l'aide de Python : **1.a.** `len(liste)` **1.b.** `liste[5]` **1.c.** `liste[4][2]`
2. Saisir et exécuter chacun des scripts suivants, expliquer ce qu'on observe et conclure sur une comparaison des deux scripts. (*On reviendra plusieurs fois sur cette question subtile !)*
2.a.

```
a = 10
b = a
a = 20
print(a,b)
```

2.b.

```
A = [10,20,30,40]
B = A
A[1] = 7
print(A,B)
```

Exercice 13.— On simule ici la génération d'une liste de bases azotées (guanine G, thymine T, adénosine A, cytosine C) sur un brin d'ADN.

1. Proposer une expression Python s'évaluant en une liste (`list`) constituée de 465 caractères égaux à `'G'`, `'T'` et `'A'` en alternance, *i.e.*

`['G','T','A','G','T','A','G','T','A',...]`.

2. Écrire un script ou une expression permettant d'engendrer la même liste que précédemment, mais en remplaçant chaque base par `'C'` toutes les cinq bases, *i.e.*

`['C','T','A','G','T','C','G','T','A','G','C','A',...]`.

Exercice 14.— Le type `tuple`, très apparenté au type `list`, n'est pas explicitement au programme, mais on le rencontre souvent en pratique. Par exemple, la fonction `divmod` appliquée à deux entiers `a` et `b` retourne le couple (ou 2-uplet) constitué du quotient et du reste de la division euclidienne de `a` par `b` :

```
In [X]: T = divmod(17,3)
In [X+1]: print(T)
(5,2)
In [X+2]: type(T)
Out[X+2]: tuple
```

1. Dans Python, affecter à une variable `T` la valeur `(1,3.1415,False,"chaîne")`.

2. Quel est son type ?

3. Que valent `len(T)`, `T[0]` et `T[3]` ?

4. Tenter de remplacer l'élément d'indice `0` par `-1`. Que se passe-t-il ? Y a-t-il des points communs avec une liste ?

5. Comment créer un 1-uplet (*n*-uplet à un seul élément), par exemple le 1-uplet contenant 3.1415 comme seul élément ? L'expression `(3.1415)` suffit-elle ?

Exercice 15.— Python peut faire des calculs avec des nombres complexes, c.à.d. des nombres de l'ensemble noté $\mathbb{C}$ en mathématiques, avec les mêmes limites d'approximations qu'avec les flottants. On se propose ici de découvrir quelques possibilités de Python en termes de calculs sur les complexes.

1. Quel est le type de l'expression `1j` ?

2. Quelles sont les types et valeurs des expressions `1j**2` et `(1.+2j)*(-2.+9j)` ?

3. Soit `z` la variable `3+4j`. Que valent `abs(z)`, `z.real`, `z.imag` et `z.conjugate()` ? Que vaut l'argument principal du complexe `z` ? On pourra se renseigner (en ligne) sur le module `numpy`.

Corrections TP 02

Correction Ex.–1

1. Très simplement, par définition de la numération à position en base 2 :

$$\boxed{\underline{101111} = 1\times 2^5 + 0\times 2^4 + 1\times 2^3 + 1\times 2^2 + 1\times 2^1 + 1\times 2^0 = 47}$$

C'est validé par Python ; essayez donc, ça ne peut pas faire de mal, même si la notation `0b...` n'est pas officiellement au programme.

2. Il faut procéder par *divisions euclidiennes* par 2 en s'arrêtant dès qu'on trouve un quotient nul :

$$55 = 27*2+1 = (13*2+1)*2+1 = ((6*2+1)*2+1)*2+1 = (((3*2+0)*2+1)*2+1)*2+1 = ((((1*2+1)*2+0)*2+1)*2+1)*2$$

En lisant les restes de droite à gauche en haut, on trouve l'écriture en binaire naturel de 55 :

$$\boxed{55 = \underline{110111}}$$

C'est validé par Python (`bin(...)` n'est pas à connaître.

3. Comme à la question 1 mais en base 8 (base dite *octale*) :

$$\boxed{(\text{octal})7413 = 7\times 8^3 + 4\times 8^2 + 1\times 8^1 + 3\times 8^0 = 3\,851}$$

On peut le vérifier en Python à l'aide de l'écriture (pas au programme) `0o7413`, qui renvoie bien 3 851.

4. Comme à la question 2 mais en base 3 :

$$63 = (21)*3+0 = (7*3+0)*3+0 = ((2*3+1)*3+0)*3+0$$

donc :

$$\boxed{63 = (\text{ternaire})2100}$$

On n'a évidemment utilisé que les chiffres de 0 à 2, comme il se doit en base 3.

Correction Ex.–2

1. Il suffit de regarder son *chiffre des unités : il vaut $\underline{1}$ si le nombre est impair et $\underline{0}$ s'il est pair.*

En effet, tout entier positif n peut s'écrire de manière unique sous la forme suivante (c'est le principe même de la numération à position en base 2, c.à.d. en binaire naturel) :

$$n = \sum_{i=0}^{N} a_i 2^i \quad \text{avec } \forall i \in \{0,\dots,N\}, \quad a_i \in \{0,1\}$$

Si $N = 0$, alors $n = a_0 \in \{0,1\}$. Dans ce cas, n vaut 0 ou 1 (ou $\underline{0}$ ou $\underline{1}$), et le critère qu'on vient de mentionner fonctionne.

Si $N \geqslant 1$, on peut isoler le chiffre des unités a_0 du reste de la somme :

$$n = \sum_{i=1}^{N} a_i 2^i + a_0$$

La somme de $i = 1$ à $i = N$ est nécessairement paire, puisque chaque terme qu'elle contient est multiple de 2. On peut donc factoriser par 2 :

$$n = 2\left(\sum_{i=1}^{N} a_i 2^{i-1}\right) + a_0 = \underbrace{2\left(\sum_{i=0}^{N} a_{i+1} 2^i\right)}_{\text{entier pair}} + a_0$$

La parité de n est donc égale à celle de a_0, qui vaut $\underline{0}$ (auquel cas n est pair) ou $\underline{1}$ (auquel cas n est impair).

2. Avec N chiffres $\underline{0}$ ou $\underline{1}$, on peut représenter tous les entiers en binaire naturel de 0 à $2^N - 1$. Ainsi :

- avec 4 chiffres, on peut écrire tous les entiers en binaire naturel de 0 à 15 ;
- avec 5 chiffres, on peut écrire tous les entiers en binaire naturel de 0 à 31 ;
- avec 6 chiffres, on peut écrire tous les entiers en binaire naturel de 0 à 63.

5 chiffres sont donc nécessaires (et suffisants) pour écrire les heures de 0 à 23. En revanche, pour les minutes et les secondes (dont le nombre est compris entre 0 et 59), il faut 6 chiffres. Au total, sur $5+6+6$=17 chiffres, on peut écrire n'importe quelle heure de la journée.

3. Comme à la question précédente :

— avec 8 chiffres, on peut écrire tous les entiers en binaire naturel de 0 à 255 ($= 2^8 - 1$) ;

— avec 16 chiffres, on peut écrire tous les entiers en binaire naturel de 0 à 65 535 ($= 2^{16} - 1$).

Le tableau ci-dessous donne par ailleurs la suite des factorielles successives :

n	1	2	3	4	5	6	7	8	9	...
$n!$	1	2	6	24	120	720	5 040	40 320	362 880	...

Avec 8 chiffres, la plus grande factorielle que l'on peut écrire est donc 5 ! = 120 (qui est la plus grande factorielle inférieure ou égale à 255). Avec 16 chiffres, c'est 8 ! = 40 320 (qui est la plus grande factorielle inférieure ou égale à 65 535).

Pour répondre à la question, vous pouvez utiliser Python comme une calculatrice.

Il faut d'abord charger le module `math` *à l'aide de l'instruction* `import`*, puis utiliser la fonction* `factorial` *du module* `math`*. On apprendra évidemment plus tard ce qu'est un module. Mais en pratique, vous pouvez faire ça :*

```
In [X]: import math # chargement du module math
In [X+1]: math.factorial(5)
Out[X+1]: 120
In [X+2]: math.factorial(6)
Out[X+2]: 720
```

Correction Ex.–3

1. On fait exactement comme en écriture décimale, avec des *retenues*, sachant que :

— $\underline{0}+\underline{0}=\underline{0}$ (pas de retenue) ;

— $\underline{0}+\underline{1}=\underline{1}+\underline{0}=\underline{1}$ (pas de retenue) ;

— $\underline{1}+\underline{1}=\underline{10}$ (« Je pose 0 et je retiens 1 ») ;

— $\underline{1}+\underline{1}+\underline{1}=\underline{11}$ (« Je pose 1 et je retiens 1 »).

Voilà l'addition telle qu'on peut la faire à la main (la première ligne correspond aux retenues) :

$$\begin{array}{cccccccc} & & {\scriptstyle 1} & {\scriptstyle 1} & & & & \\ & & 1 & 0 & 1 & 1 & 0 & 0 \\ + & & & 1 & 1 & 0 & 0 & 1 \\ \hline = & 1 & 0 & 0 & 0 & 1 & 0 & 1 \end{array}$$

Donc :

$$\boxed{\underline{10\,1100}+\underline{1\,1001}=\underline{1000101}=69}$$

et c'est vrai, car :

$$\underbrace{\underline{10\,1100}}_{=44}+\underbrace{\underline{1\,1001}}_{=25}=69$$

2. La réponse est :

décimal	binaire naturel
0	$\underline{0}$
1	$\underline{1}$
2	$\underline{10}$
3	$\underline{11}$
4	$\underline{100}$
5	$\underline{101}$
6	$\underline{110}$
7	$\underline{111}$
8	$\underline{1000}$
9	$\underline{1001}$
10	$\underline{1010}$
11	$\underline{1011}$
12	$\underline{1100}$
13	$\underline{1101}$
14	$\underline{1110}$
15	$\underline{1111}$
16	$\underline{10000}$

Il faut bien étudier comment on passe d'un nombre au suivant (avec les retenues, etc.), afin de se forger une bonne intuition de la numération binaire naturelle. En particulier, lorsqu'on ajoute 1 à un nombre de la forme $\underline{111\ldots111}$ constitué uniquement de 1, on remplace tous les $\underline{1}$ par des $\underline{0}$ et on ajoute un $\underline{1}$ à gauche. C'est exactement comme ajouter 1 à un nombre de la forme 999…999 en écriture décimale (99 999 999 + 1 = 100 000 000).

Correction Ex.–4

1. En base 10, rajouter un 0 à droite, c.à.d. décaler tout vers la gauche, revient à multiplier par 10. En base 2, *décaler à gauche/-rajouter un 0 à droite revient à multiplier par 2*.

C'est très simple à montrer. Soit un entier naturel quelconque n, que l'on peut écrire en binaire naturel :

$$n = \underline{a_N \dots a_1 a_0} = a_N \times 2^N + \dots + a_1 \times 2^1 + a_0 \times 2^0 = \sum_{i=0}^{N} a_i 2^i$$

Soit n' le nombre qu'on obtient en ajoutant un zéro à droite dans l'écriture binaire naturelle de n :

$$n' = \underline{a_N \dots a_1 a_0 0} = \underbrace{a_N \times 2^{N+1} + \dots + a_1 \times 2^2 + a_0 \times 2^1}_{=2\left(a_N \times 2^N + \dots + a_1 \times 2^1 + a_0 \times 2^0\right) = 2n} + \underbrace{0 \times 2^0}_{=0} = 2n$$

comme on s'en doutait.

2. Soit :

$$n = \underline{a_N \dots a_1 a_0} = a_N \times 2^N + \dots + a_1 \times 2^1 + a_0 \times 2^0 = \sum_{i=0}^{N} a_i 2^i$$

un entier naturel quelconque. On peut le supposer supérieur ou égal à 2, faute de quoi son décalage à droite fait disparaître tous ses chiffres (0 et 1 n'ayant qu'un chiffre en binaire naturel). Autrement dit, on peut supposer $N \geqslant 1$.

Soit n' l'entier obtenu en décalant tous les chiffres de n vers la droite. Autrement dit :

$$n' = \underline{a_N \dots a_1} = a_N \times 2^{N-1} + \dots + a_1 \times 2^0 = \sum_{i=1}^{N} a_i 2^{i-1}$$

On a donc :

$$2n' = 2\sum_{i=1}^{N} a_i 2^{i-1} = \sum_{i=1}^{N} a_i 2^i \quad \text{(attention, il n'y a pas de terme } i = 0\text{)}$$

d'où :

$$n = 2n' + a_0$$

Or, a_0 (chiffre de poids faible de n) vaut 0 ou 1. On reconnaît donc exactement la *division euclidienne de n par 2, où n' est le quotient et a_0 le reste*.

Le résultat du décalage vers la droite des chiffres de n donne donc en binaire naturel le quotient de la division euclidienne de n par 2.

⋆⋆⋆

Les opérations de décalage de bits existent en Python (mais c'est hors-programme). Elles se notent `<<` (décalage vers la gauche) et `>>` (décalage vers la droite). Ainsi, `13<<3` renvoie un résultat égal à trois décalages successifs des bits de 13 (écrit en binaire naturel) vers la gauche, autrement dit à une multiplication par $2^3 = 8$. On obtient bien $13 \times 8 = 104$:

```
In [X]: 13<<3
Out[X]: 104
```

De même, en faisant `13>>2`, on décale tous les bits de 13 (écrit en binaire naturel) deux fois vers la droite, ce qui revient à faire deux divisions euclidiennes successives par 2. On obtient donc 3, vu que $13 = 4 \times 3 + 1$:

```
In [X]: 13>>2
Out[X]: 3
```

Cette opération de décalages de bits dans un sens ou dans l'autre a surtout du sens au niveau des composants mêmes de l'ordinateur*, dans les mémoires appelées* registres*. Elles permettent de réaliser des opérations directement sur les bits* physiques *(transistors). En fait, l'expérience montre même que les opérations de la forme `n<<1` et `n>>1` (avec `n` entier) sont un peu plus rapides que les opérations `n*2` et `n//2`, qui retournent le même résultat.*

Correction Ex.–5

1. Sachant que le chiffre hexa. F vaut 15 et que le chiffre hexa. D vaut 13 :

$$\boxed{\text{(hexa.)}D2F = 13 \times 16^2 + 2 \times 16^1 + 15 \times 16^0 = 3\,375}$$

C'est facile à vérifier avec Python, en mode calculatrice.

On peut directement utiliser le préfixe `0x` (hors-programme)

```
In [X]: 0xD2F
Out[X]: 3375
```

2. Il faut procéder par *divisions euclidiennes* par 16 en s'arrêtant dès qu'on trouve un quotient nul :

$$244 = 15 * 16 + 4$$

Le chiffre qui correspond à 15 en hexadécimal est F, donc :

$$\boxed{244 = (\text{hexa.})F4}$$

La commande hors-programme `hex` *permet de le vérifier :*

```
In [X]: hex(244)
Out[X]: '0xf4'
```

3. La *conversion de l'hexadécimal vers le binaire naturel* est très simple, et c'est une des raisons pour lesquelles on écrit souvent les chiffres d'un nombre en binaire naturel *par groupes de quatre* séparés par une espace. C'est en réalité lié au fait que $2^4 = 16$, autrement dit qu'avec des nombres binaires naturels à quatre chiffres, on peut écrire tous les entiers de 0 à 15. C'est plus simple si on reprend l'exemple de la question 1 :

$$D2F_{16} = 13 \times 16^2 + 2 \times 16^1 + 15 \times 16^0$$

Chacun des nombres 13, 2 et 15 (soit D, 2 et F en hexadécimal) peut être écrit en binaire naturel avec quatre chiffres (et c'est d'ailleurs vrai pour tout entier compris entre 0 et 15, c.à.d. pour tous les symboles hexadécimaux) :

$$(\text{hexa.})D2F = \underbrace{13}_{=\underline{1101}} \times 16^2 + \underbrace{2}_{=\underline{0010}} \times 16^1 + \underbrace{15}_{=\underline{1111}} \times 16^0$$

En *juxtaposant* à la suite les écritures en binaire naturel de 13, 2 et 15, c.à.d. de D, 2 et F, on obtient un nouveau nombre binaire naturel à $3 \times 4 = 12$ chiffres dont on vérifie facilement après calcul qu'il est égal à 3 375, soit $D2F$ en hexa. :

$$\overbrace{\underline{1101}}^{=D}\ \overbrace{\underline{0010}}^{=2}\ \overbrace{\underline{1111}}^{=F} = \ldots = 3\,375 = (\text{hexa.})D2F$$

C'est en fait tout à fait général et assez facile à démontrer (je vous laisse essayer) : *pour convertir un nombre de l'hexadécimal vers le binaire naturel, on convertit chacun de ses chiffres en binaire naturel (avec obligatoirement quatre chiffres, quitte à laisser des $\underline{0}$ à gauche), puis on juxtapose les nombres binaires naturels obtenus*.

Par conséquent, *un même nombre est quatre fois plus long (en termes de nombre de chiffres) si on l'écrit en binaire naturel plutôt qu'en hexadécimal*. Ce qu'on économise en termes de nombres de symboles disponibles (2 ou 16) se répercute évidemment sur le nombre de chiffres de chaque nombre, avec éventuellement une (petite) économie si le premier chiffre de l'écriture hexadécimale est inférieur ou égal à 7 (auquel cas son écriture binaire utilise trois chiffres ou moins au lieu de quatre). Bref, la clef hexadécimale de l'énoncé, qui utilise 26 chiffres hexadécimaux, s'écrirait avec $26 \times 4 =$ *104* chiffres en binaire naturel. Ce serait :

$$(\text{hexa.})\overbrace{D33645E44FA643BC44F9FA79A6}^{\text{26 chiffres héxadécimaux}} = \underbrace{\overbrace{1101}^{=D}\ \overbrace{0110}^{=3}\ldots\overbrace{1010}^{=A}\ \overbrace{0110}^{=6}}_{\text{104 chiffres en binaire naturel}}$$

En décimal, Python (`0xD33645E44FA643BC44F9FA79A6`) nous révèle que :

$$(\text{hexa.})\overbrace{D33645E44FA643BC44F9FA79A6}^{\text{26 chiffres héxadécimaux}} = \overbrace{16\,733\,938\,975\,090\,512\,777\,902\,303\,115\,686}^{\text{29 chiffres décimaux}}$$

L'écriture décimale de la clef a une longueur de 29 chiffres, ce qui est un peu plus qu'en hexadécimal. L'idée de l'écriture en hexadécimal d'une clef Wifi est donc de *réduire le nombre de chiffres (et donc le risque d'erreurs lors de la saisie)* tout en augmentant la lisibilité (quelques lettres au milieu des chiffres arabes usuels constituent des jalons permettant mentalement une meilleure lisibilité du nombre). Ça ne compromet pas la sécurité pour autant, puisque le nombre total de clefs possibles ($26^{16} = 43\,608\,742\,899\,428\,874\,059\,776$, c.à.d. beaucoup) est indépendant de la base choisie.

Correction Ex.–6

1. On va avoir besoin d'utiliser la *fonction logarithme népérien*, qu'il faut charger depuis le module `numpy` par exemple (ou depuis le module `math`, qui est contenu dans `numpy`). Attention, la fonction ln se note `log` et pas `ln` (tandis que le logarithme décimal log se note `log10` en Python).

Le script suivant affiche le résultat demandé :

```
import numpy #pour disposer des fonctions mathématiques usuelles}
print(numpy.log(10.))
```

La valeur `2.302585092994046` s'affiche alors dans le shell.

Si on veut plutôt utiliser le module `math` :

```
import math #pour disposer des fonctions mathématiques usuelles}
print(math.log(10.))
```

Si on veut éviter de préfixer par `numpy` (ce que je ne recommande *pas*), on peut ne charger *que* la fonction `log` du module :

```
from numpy import log #pour ne charger que la fonction log
print(log(10.)) #plus besoin de préfixer par numpy
```

Si on veut charger tout le module mais qu'on a la flemme de tout préfixer (pas recommandé non plus), il faut faire :

```
from numpy import * #pour charger TOUTES les fonctions de numpy
print(log(10.)) #plus besoin de préfixer par numpy non plus
```

Enfin, si on trouve que `numpy` est trop long, on peut l'*aliaser* en `np` :

```
import numpy as np
print(np.log(10.)) #le préfixe est très raccourci CONSEILLE!!
```

2. On va avoir besoin d'utiliser la *fonction exponentielle*, qu'il faut charger depuis le module `math` par exemple (ou depuis le module `numpy`), comme précédemment.

Le script suivant affiche le résultat demandé :

```
import numpy as np
x = 0.5
print(np.exp(-1./(1.-x**2)))
```

La valeur `0.26359713811572677` s'affiche alors dans le shell.

3. On a cette fois besoin de la *fonction factorielle*. Une recherche sur Internet montre qu'elle est définie dans le module `math` sous le nom `factorial`. On peut donc calculer :

```
import math
print(math.factorial(10)//math.factorial(5)**2)
```

On obtient le résultat 252. *Attention*, le résultat est supposé être un *entier* : il faut *impérativement faire une division entière (double-slash //), c.à.d. calculer le quotient (entier) de la division euclidienne*. Si on ne met qu'un seul trait de fraction `/`, le résultat est automatiquement converti en *flottant* (on obtient `252.0`). C'est *important*.

Sur le même modèle, on calcule le coefficient binomial demandé et on conclut avec la comparaison en effectuant un quotient qui est effectivement proche de 1.

On a aussi besoin de la constante π dont une valeur approximative est disponible dans le module `math` (par exemple, ou dans `numpy`) et de la fonction « racine carrée » `math.sqrt`. Bref, une possibilité est :

```
import math
q = math.factorial(100)//math.factorial(50)**2
q2 = 2**100/math.sqrt(50*math.pi)
print(q/q2)
```

4. On est tenté (mais c'est vrai que ce n'est pas obligatoire) de définir une variable `lambda` égale à `0.2`. Ça ne marche pas, parce que `lambda` *est un mot réservé du langage* (il sert à définir ce qu'on appellera plus tard une fonction anonyme). On obtient en effet un message d'erreur du type :

```
lambda = 0.2
  File "<stdin>", line 1
    lambda = 0.2
           ^
SyntaxError: invalid syntax
```

Comme Python est *sensible à la casse* (c.à.d. aux majuscules et minuscules), on peut définir une variable `Lambda` avec majuscule, par exemple (ou tout autre nom intelligible), au lieu de `lambda`.

```
Lambda = 0.2 #lambada c'est bien aussi
print(2.*math.pi/Lambda)}
```

On obtient `31.41592653589793`.

Correction Ex.–7

1. Notons a_0 et b_0 les valeurs initiales des variables `a` et `b`. Les contenus en mémoire sont successivement :

contenu de la variable `a`	contenu de la variable `b`
a_0	b_0
affectation `a = a+b` :	
a_0+b_0	b_0
affectation `b = a-b` :	
a_0+b_0	a_0
affectation `a = a-b` :	
b_0	a_0

On constate donc que cette suite d'affectations a *échangé les valeurs des variables* `a` *et* `b`.

2. On peut par exemple saisir et exécuter le script suivant :

```
a = 4.125 # ou n'importe quel autre flottant
b = -3.88 # ou n'importe quel autre flottant
print("valeurs initiales de a et b :",a,b)
a = a+b
b = a-b
a = a-b
print("valeurs finales de a et b :",a,b)
```

3. On a ici *échangé les valeurs de* `a` *et* `b` *sans variable auxiliaire*. On n'a pas utilisé d'espace-mémoire autre que celui qui était déjà alloué à `a` et `b`. On sait en effet qu'en général (cf cours), la méthode pour échanger les contenus de deux variables `a` et `b` est :

```
c = a #création d'une variable intermédiaire c de stockage de la valeur initiale de a
a = b #écrasement de a et remplacement par la valeur de b
b = c #récupération de la valeur initiale de a
```

On peut donc parler de *calcul sans mémoire additionnelle*, puisque la méthode présentée dans l'exercice permet théoriquement d'économiser l'espace-mémoire qu'occuperait une variable de stockage annexe `c`. En pratique, on n'est vraiment pas à ça près et il faudrait faire preuve d'une avarice invraisemblable pour en arriver là. Ce n'est donc qu'un prétexte à faire un exercice permettant de s'approprier les mécanismes d'affectation. Ceci dit, vous pouvez jeter un œil à l'article suivant, tiré de l'*excellente chronique mensuelle « Logique et calcul » de Jean-Paul* DELAHAYE dans la revue Pour la Science (N^{o}404, 1999) :

`https://www.pourlascience.fr/sd/mathematiques/le-calculateur-amnesique-2588.php`

4. En Python, la syntaxe suivante (*spécifique à Python*) marche aussi, et est très commode :

```
a,b = b,a #échange les valeurs de a et b
```

Correction Ex.–8 *En cas de doute sur ce qui suit, vous pouvez utiliser la fonction* `type` *qui vous donne le type de n'importe quelle expression ou variable. Par exemple,* `type(42)` *retourne* `int` *(entier).*

1. Dans `91//2`, on reconnaît l'opération de *division entière* (quotient de la division euclidienne) de `91` par `2`, qui donne `45`. D'après le cours, l'opérateur binaire `!=` est l'*opérateur « différent de »*, qui retourne un *booléen*.

L'expression `44 != 91//2` *est donc de type booléen*, et sa valeur est `True` car 44 et 45 sont différents (« $44 \neq 45$ » est vrai). Si vous n'êtes pas convaincus, saisissez `44 != 91//2` dans le shell (cette remarque vaut d'ailleurs pour toute la suite de l'exercice).

2. On reconnaît la structure d'une liste `list` car une certaine expression est encadrée de `[` et `]`.

Il s'agit en fait d'une *liste construite par compréhension* que l'on revisitera plus tard. Il s'agit de la liste des restes des divisions euclidiennes des i^2+11 par 7, pour i compris entre 1 inclus et 10 exclu [1]. Or :

$$1^2+11=12=1\times 7+5 \qquad 2^2+11=15=2\times 7+1 \qquad 3^2+11=20=2\times 7+6 \quad \text{etc.}$$

Après calcul, cette liste est donc la liste `[5,1,6,6,1,5,4,5,1]` (vous pouvez vérifier avec Python).

3. Comme le slash `/` est simple, cette division de deux entiers est un *flottant* égal à `2.0`. Pour obtenir un résultat entier (`2`), il aurait fallu faire une division entière (`//`)

4. On reconnaît la *concaténation (opérateur* `+`*) de deux chaînes de caractères*, qui donne donc une *chaîne de caractères* bien connue de tous les élèves de CPGE qui ont lu les œuvres du programme (de français-philosophie).

Remarquer qu'on peut encadrer une chaîne de caractère par des apostrophes (`' '`) (guillemets simples [2]) ou des guillemets anglo-saxons (`" "`) (guillemets doubles [3]). Si on veut que la chaîne contienne le caractère apostrophe ', il faut mettre des guillemets doubles (pour distinguer l'apostrophe des délimiteurs de la chaîne) ou bien utiliser le caractère `\'`.

1. Le premier indice est toujours inclus et le dernier est toujours exclu.
2. mauvaise traduction littérale de *single quotes*
3. mauvaise traduction littérale de *double quotes*

5. Le `[3]` indique qu'il s'agit du caractère d'indice `3` (c.à.d. `n`) de la chaîne précédente. L'expression est donc de type *chaîne de caractères*, et c'est la chaîne à un seul caractère `"n"`.

6. `[8,9,10]` est la liste des trois entiers `8`, `9` et `10`. Ajouter `[2]` à droite signifie qu'on extrait de cette liste l'élément d'indice 2, à savoir `10` (*en Python, les indices commencent à 0*, rappelons-le). C'est comme si on avait affecté la valeur `[8,9,10]` à une variable `L` (de type liste, donc), et qu'on s'intéressait au type de l'expression `L[2]`.

`[8,9,10][2]` *est donc un entier égal à* `10`. C'est certes un peu tordu et peu lisible, mais c'est parfaitement légal en Python.

7. `int` est ici un opérateur de *conversion* : il permet de convertir le flottant `5.6` en l'entier `5`.
Le résultat est donc l'*entier* `5`.

8. C'est le contraire de la question 7 : l'opérateur `float` permet de convertir l'entier `5` en le flottant `5` (ou `5.0`).
Le résultat est donc le *flottant* `5.0`.

9. `str(455.8)` est le résultat de la *conversion* du flottant `455.8` en chaîne de caractères, c.à.d. la chaîne de caractères `"455.8"`. L'expression `len(str(455.8))` est donc la longueur de cette chaîne, c.à.d. l'*entier* `5`.

Correction Ex.–9 Il suffit de faire un *test avec des valeurs numériques particulières* de `a`, `b` et `c`, par exemple `2`, `3` et `4`.
On calcule (mathématiquement) :

$$\left(2^3\right)^4 = 8^4 = 4\,096$$

et

$$2^{(3^4)} = 2^{81} = 2\,417\,851\,639\,229\,258\,349\,412\,352$$

Or, si on saisit `2**3**4` dans le shell, on obtient `2417851639229258349412352`. `a**b**c` *est donc égal à* `a**(b**c)`.
Ceci dit, en pratique et pour des raisons de lisibilité, il vaudrait mieux mettre des parenthèses. Ça ne coûte pas cher et c'est plus clair.
Pour traiter le cas de `//`, essayer avec $a = 12$, $b = 3$, $c = 2$.

Correction Ex.–10
1.a. Assez logiquement, l'opérateur `not` est prioritaire sur `and` et `or`, et `and` est prioritaire sur `or`. L'expression est donc égale à :

$$\underbrace{\texttt{False or } \underbrace{\texttt{False and } \underbrace{\texttt{not True}}_{\texttt{False}}}_{\texttt{False}}}_{\texttt{False}}$$

Le résultat est donc `False`, ce que confirme le shell.
C'est bien joli, les règles de précédence*, mais ce serait quand même plus clair avec des parenthèses. Ce n'est pas obligatoire, mais l'expression* `False or (False and (not True))` *est quand même plus lisible.*
En réalité, `not True` *n'est même pas évalué, à cause de la* paresse de l'opérateur `and`*. En effet,* `False and X` *est nécessairement égal à* `False`*, quel que soit* `X` *(*`True` *ou* `False`*). Quand Python lit* `False and...`*, il ne lit même pas ce qu'il y a après* `and`*, vu que c'est inutile (le résultat sera* `False`*). On peut s'en assurer en remplaçant ici* `not True` *par n'importe quoi, y compris une expression qui n'a pas de sens. Voir le script* `Exercice 3.5.py`.
1.b. Le parenthésage complet correct de cette expression est :

```
True or (not(True) and False)
```

la valeur est donc `True`. A noter que (mauvaise application des règles de précédence)

```
(True or not(True)) and False
```

vaut `False`.
1.c. Ici, c'est un peu plus délicat. Les *opérateurs de comparaison* (`==` et `!=`) sont prioritaires sur `not`, `and` et `or` (assez logiquement aussi, d'ailleurs).
Sachant que $17 \times 19 = 323$, l'expression de l'énoncé est équivalente à :

$$\underbrace{\underbrace{\texttt{323 != 324}}_{\texttt{True}} \texttt{ or not } \underbrace{\texttt{True == False}}_{\texttt{False}}}_{\texttt{True}}$$

Le résultat vaut donc `True`.
Mêmes remarques :

- L'usage de parenthèse est quand même recommandé pour des raisons de lisibilité.
- *Vu que* `True or X` *est vrai quel que soit* `X`*, ici* `not True == False` *n'est en réalité pas évalué par Python. C'est lié à la paresse de l'opérateur* `or`.

2. Oui, *manifestement* `1000 < 10 < 100` *a du sens en Python*, puisque Python retourne un résultat (`False`). C'est en fait équivalent à l'expression `1000 < 10 and 10 < 100`, qui est effectivement égale à `False` car l'expression `1000 < 10` vaut `False`.

De même pour l'expression `1000 > 10 < 100`, qui a du sens, et qui est équivalente à `1000 > 10 and 10 < 100`, ce qui vaut donc `True`.

L'expression `1000 < 10 < 1/0` est équivalente à `1000 < 10 and 10 < 1/0`. Or, `1000 < 10` vaut `False` : la suite n'est même pas évaluée, par paresse de `and`. Le résultat a du sens et vaut donc `False`. Ça a du sens en Python, mais pas en mathématiques !

3. C'est étonnant, mais les concepteurs de Python ont programmé une *relation d'ordre* sur l'ensemble {vrai, faux}. En effet, on constate que `True > False` retourne `True` : on en déduit que non seulement l'expression `True > False` a du sens, mais aussi que `True` est plus grand que `False`. Je ne connais pas l'utilité de cette relation d'ordre, mais elle existe.

4. L'expression booléenne demandée (lourde) est simplement :

```
("a" in mot) and ("e" in mot) and ("i" in mot) and ("o" in mot)
                                                and ("u" in mot) and ("y" in mot)
```

Pour culture, « Portez ce vieux whisky au vieux juge blond qui fume » est une des phrases françaises les plus courtes parmi celles qui contiennent toutes les lettres de l'alphabet (c'est un pangramme) :

`https://fr.wikipedia.org/wiki/Pangramme`

Quant à « A man, a plan, a canal : Panama. », c'est un palindrome (se lit à l'endroit comme à l'envers) en anglais qui, cerise sur le gâteau, n'admet que des « a » pour voyelles. Il y a aussi évidemment de très jolis palindromes en français. Mon préféré est « Tu l'as trop écrasé, César, ce Port-Salut. » :

`https://fr.wikipedia.org/wiki/Palindrome`

Correction Ex.–11

1.a. On trouve que `len(chaine)` vaut 30. C'est tout simplement le nombre de caractères de la chaîne de caractères `chaine`. Les espaces et le point final sont des caractères comme les autres et comptent pour un caractère.

1.b. `chaine[5]` est le caractère d'indice 5 de la variable `chaine`, c.à.d. le sixième (*en Python, les indices commencent à* 0). On voit bien que le sixième caractère de `chaine` est effectivement `"u"`.

1.c. `chaine[3:16]` est la sous-chaîne des caractères dont les indices commencent à 3 (inclus) et finissent à 16 (*exclu*). C'est donc la chaîne `"neutron est a"`.

1.d. `chaine[3:16:4]` est la sous-chaîne des caractères dont les indices commencent à 3 (inclus) et finissent à 16 (exclu), par pas de 4. C'est donc la chaîne `"nrea"`, constituée des caractères d'indices 3, 7, 11 et 15.

1.e. `chaine[-1]` est le caractère `"."`, qui est le *dernier caractère de la chaîne*.

1.f. `chaine[-2]` est le caractère `"e"`, qui est l'*avant-dernier caractère de la chaîne*.

1.g. `chaine[-3]` est le caractère `"d"`, qui est le *troisième caractère de la chaîne à partir de la fin*. Il apparaît donc que *les indices négatifs sont autorisés en Python, et permettent de compter à partir de la fin* (*idem* pour les listes), ce qui est parfois très pratique.

1.h. D'après le cours, `chaine[10:]` est la sous-chaîne qui commence à partir de l'indice 10 (inclus). C'est confirmé par l'expérience, puisqu'on obtient `" est aussi une onde."` (qui commence par une espace).

1.i. De même, `chaine[:10]` est la sous-chaîne qui commence au début et se termine à l'indice 10 (*exclu*). On obtient bien `'Le neutron'`.

1.j. Enfin, `chaine[:]` semble égal à la chaîne `chaine` elle-même. Ça a l'air idiot, mais en fait, `chaine[:]` n'est pas vraiment identique à `chaine` : il faut voir `chaine[:]` comme le *contenu* de `chaine`.

Tous ces mécanismes (slicing) d'extraction de caractère ou de sous-chaîne sont en effet communs aux chaînes de caractères, aux listes, ainsi qu'aux n-uplets (c.f. exercice 14).

1.k. `chaine[::-1]` est la chaîne `chaîne` *renversée* (`".edno enu issua tse nortuen eL"`). Il faut comprendre ça comme « Tous les caractères de `chaine`, du début à la fin, par pas de `-1` ». C'est très commode.

2. On trouve 26 : *le caractère apostrophe* `"'"` *compte pour un seul caractère*. En fait, ici, le backslash n'est pas nécessaire car on a délimité la chaîne à l'aide de guillemets doubles. Mais si on l'avait délimité par des guillemets simples (`'C\'est trop un truc de ouf.` il aurait été nécessaire.

3. Il suffit de doubler le '\' comme dans `chemin = "C:\\Users\\"`.

4. Le dernier caractère de la deuxième est un « retour chariot », code ASCII 13, dont la fonction est de passer à la ligne. (Cette terminologie est issue des usages sur les machines à écrire mécaniques.) C'est le caractère de fin de ligne dans certains fichiers textes (la convention de fin de ligne est spécifique au système d'exploitation, c'est un problème lorsqu'on échange des fichiers textes d'une machine à l'autre).

Correction Ex.–12

1.a. On trouve 9. En effet, `liste` contient les entiers 1, 2, 3, 13, 14, 15, 16, mais aussi deux listes (`[4,5,6,7,8]` et `[9,10,11,12]`) qui comptent chacune pour un élément.

1.b. `liste[5]` est le sixième élément (indice 5) de `liste`, c.à.d. l'entier 13.

1.c. `liste[4][2]` est l'élément d'indice 2 de la liste `liste[4]`, égale à `[9,10,11,12]`, c.à.d. l'entier 11. On peut ranger ce qu'on veut dans une liste, y compris d'autres listes !

2.a. L'exécution affiche `20` et `10`, exactement comme on pouvait s'y attendre.
2.b. On pourrait s'attendre à obtenir `[10,7,30,40]` et `[10,20,30,40]`, comme à la question précédente. On obtient en fait deux fois `[10,7,30,40]` !

En fait, l'instruction `B = A` n'a *pas* créé un nouvel objet `B` : elle n'a fait qu'assigner l'identifiant `B` à la liste déjà identifiée par `A`.

Ce phénomène de non-duplication d'objets est source de très nombreuses erreurs dans les manipulations de listes, notamment lorsque celles-ci sont passées en argument dans une fonction. C'est aussi un phénomène que l'on peu mettre à profit.

Correction Ex.–13
1. 465 est un multiple de 3 :

$$\frac{465}{3} = 155$$

Il suffirait donc de concaténer 155 fois la liste `['G','T','A']`, ce qui est autorisé en Python par l'opérateur `*` : « multiplier » une liste par un entier `n` revient à concaténer `n` fois la liste. La liste demandée correspond par exemple à l'expression suivante :

```
['G','T','A']*155
```

2. On peut *s'y prendre en plusieurs temps*. On peut par exemple commencer par affecter provisoirement la valeur de la liste de la question 1 à une variable `L` :

```
L = ['G','T','A']*155
```

On peut ensuite *modifier certains éléments cette liste* `L` : il suffit de remplacer les éléments d'indices `0`, `5`, `10`,..., `460` (il n'y a pas d'élément d'indice `465`, car la liste `L` a 465 éléments, et que le premier porte l'indice `0`) par le caractère `'C'`.

La sous-liste des éléments dont l'indice est multiple de 5 est simplement `L[::5]` (« tous les éléments, depuis le début, par pas de 5, jusqu' à la fin »). Cette sous-liste comporte $465/5 = 93$ éléments (c'est facile à vérifier avec la fonction `len`).

Il suffit donc de remplacer `L[::5]` par la liste contenant 93 fois l'élément `'C'` :

```
L[::5] = ['C']*93
```

La liste `L` est alors la liste demandée (vous pouvez vérifier).

Autre méthode plus laborieuse, à éviter : il est clair que la liste recherchée est périodique, de période $3 \times 5 = 15$ (3 et 5 étant premiers entre eux). Sachant que $465/15 = 31$, il suffit de concaténer 31 fois la liste

```
['C','T','A','G','T','C','G','T','A','G','C','A','G','T','A']
```

(calculée à la main) :

```
['C','T','A','G','T','C','G','T','A','G','C','A','G','T','A']*31
```

est la liste demandée.

Une autre méthode consisterait à utiliser une boucle inconditionnelle `for`.

Correction Ex.–14
1. En console,

```
T = (1,3.1415,False,"chaîne")
```

Remarquer que les parenthèses ne sont pas obligatoires.
2. `type(T)` en console retourne `<class 'tuple'>`
3. `len(T)` : 4, `T[0]` : `1` et `T[3]` est le booléen `False`
4. En console, on tente `T[0]=-1`. Ce qui donne

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Les points communs avec une liste sont la lecture d'éléments du tuple. La différence fondamentale est qu'on ne peut modifier un tuple, comme une chaïnne de caracatères.
5. Pour obtnir un tuple à 1 élément, il suffit de placer une virgule finale,

```
U = 3.14159,
```

ou

```
U = (3.14159,)
```

Une erreur extrêmement commune, très difficilement décelable, est le remplacement d'un . décimal par une , : `pi = 3,141592` *donne un tuple.*

Correction Ex.–15

1. On trouve (à l'aide de l'instruction `type`) que l'expression `1j` est de type `complex`. Les complexes existent en Python, sans recours à un module.

`1j` est naturellement le nombre que vous noteriez i en mathématiques. Python respecte les conventions de physiciens, qui notent j le nombre i (ici, pour éviter les confusions avec les indices couramment notés `i`).

2. On trouve que ces deux expressions sont de type `complex`, et valent respectivement `(-1+0j)` (comme prévu : $i^2 = -1 = -1+0\times i$) et `(-20+5j)` (*idem* : $(1+2i)(-2+9i) = \ldots = -20+5i$).

3. On trouve que :

— `z.abs` vaut `5.0` : c'est le *module* de `z`. En effet, $|3+4i| = \sqrt{3^2+4^2} = 5$;

— `z.real` vaut `3.0` : c'est la *partie réelle* de `z` ;

— `z.imag` vaut `4.0` : c'est la *partie imaginaire* de `z` ;

— `z.conjugate()` vaut `(3-4j)` : c'est le *complexe conjugué* de `z`.

Pour l'argument, il faut faire une petite recherche en ligne (ce qui doit faire partie de votre savoir-faire d'après le programme). On trouve que le module `numpy` contient une fonction `angle` qui retourne l'argument d'un complexe, pris en radians dans l'intervalle $]-\pi,\pi]$. Ainsi, `numpy.angle(z)` vaut environ `0.9272952180016122`, qui vaut approximativement $\arg(3+4i) = \arctan(4/3)$.