COURS NOMBRES et OPERATIONS

1 – NOMBRES à VIRGULES :

Nombres à virgule

Afficher un nombre à virgule

En Python, les nombres à virgules (ou nombres décimaux) ne s'écrivent pas avec une "virgule" mais avec un "point", comme le font les anglais ou les américains. Ainsi, si on veut afficher le nombre "3,14" on va écrire le programme suivant :

```
print(3.14)
```

```
3.14
```

Déclarer un nombre à virgule

Pour affecter à une variable une valeur décimale (c'est-à-dire non entière), ou pour faire des calculs, on fait comme pour les entiers.

```
prixJeu = 29.99
prixConsole = 299
print(prixJeu + prixConsole - 49.95)
```

```
279.04
```

Attention de bien utiliser un point et pas une virgule quand vous écrivez des nombres « à virgule », sinon vous aurez des surprises. Regardons simplement quelques exemples :

```
print((1,5) + 2)
```

```
TypeError: can only concatenate tuple (not "int") to tuple
```

Autre cas possible :

```
print(1,5 + 2)
```

```
1 7
```

Il faut donc toujours utiliser un point pour les "nombres à virgule" !

2 – FAIRE DESDIVISIONS :

Comme en mathématiques, si on veut diviser deux nombres, on les sépare par le symbole `/`

Il est très simple de diviser entre eux des nombres à virgule ou des entiers, on obtiendra toujours un nombre à virgule comme résultat :

```
print(10.5 / 3.5)
print(10 / 3)
print(10 / 2)
print(10 / 4.5)
```

```
3.0
3.3333333333333335
5.0
2.2222222222222223
```

3 – LECTURE :

Lecture de nombres à virgule

Pour lire un nombre à virgule on utilise le code suivant :

```
nombre = float(input())
print(nombre * 2)
```

```
10.5
```

```
21.0
```

Cela ressemble donc beaucoup à la lecture d'un nombre entier, on utilise simplement `float(input())` au lieu de `int(input())`.

Remarque: Le mot "float" vient de l'anglais "floating-point" qui signifie "à virgule flottante".

LECTURE DE DECIMAUX : ERREUR POSSIBLE

Que se passe-t-il si on essaie de lire un entier alors que le nombre qu'on nous donne est un nombre décimal ?

```
x = int(input())
print(x)
```

```
12.34
```

```
ValueError: invalid literal for int() with base 10: '12.34'
```

Le programme s'attendait donc à avoir un entier mais il y avait un nombre décimal, ce qui a provoqué une erreur.

4 – PRECISION DES CALCULS :

Des calculs approchés

Quand on travaille avec des nombres à virgule, le résultat affiché n'est pas toujours exact, car seulement 17 chiffres (avant ou après la virgule) seront conservés au maximum :

```
print(2 / 3)
```

```
0.6666666666666666
```

Le fait que les nombres puissent être arrondis donne parfois lieu à des résultats surprenants, comme l'illustre l'exemple suivant :

```
print(1.0 - 0.00000000000000000000001)
```

```
1.0
```

Au contraire, lorsqu'on travaille avec des nombres entiers, les résultats ne sont jamais arrondis. En particulier, les nombres peuvent devenir aussi grands qu'on le souhaite !

```
print(12345678987654321 * 98765432101234567 89)
```

```
1219326319798811157855509831126352 69
```

Dès qu'on utilise une division, on obtient forcément un nombre à virgule en résultat et cela peut parfois être surprenant :

```
grosNombre = 1000 * 1000 * 1000 * 1000 * 1000 * 1000
nombre = grosNombre + 1
print(nombre)
nombre = (grosNombre + 1) / 1
print(nombre)
nombre = (grosNombre + 1) / 1 - grosNombre
print(nombre)
```

```
1000000000000000001
1e+18
0.0
```

Le "1e+18" ci-dessus est en fait égal au nombre constitué d'un "1" suivi de 18 "0". Cette notation est expliquée plus en détail au cours suivant.

Mais que s'est-il passé ? Quand on a divisé par 1 on est passé d'un nombre entier (avec un nombre de chiffres non-limité) à un nombre à virgule (qui ne garde que 17 chiffres) et donc le chiffre des unités a été "oublié". Même quand on soustrait le nombre *grosNombre* on ne peut récupérer ce chiffre des unités et le résultat est donc égal à 0.0 !

Un exemple étonnant

Considérons l'exemple suivant :

```
prixJeu = 29.99
prixConsole = 299
print(prixJeu + prixConsole - 49.90)
```

```
279.09000000000003
```

Le résultat "mathématique" serait bien sur "279.09" alors pourquoi y a-t-il ce "3" tout à la fin ? La méthode utilisée par l'ordinateur pour stocker en mémoire les nombres à virgules ne permet pas de stocker de manière exacte tout nombre à virgule. Ainsi de très légères erreurs peuvent apparaître pour les tout derniers chiffres.

Pas de tests d'égalité

Considérons le programme suivant :

```
if 0.1 + 0.2 == 0.3:
   print("Exact")
else:
   print("Approché")
print(0.1 + 0.2 - 0.3)
```

```
Approché
5.551115123125783e-17
```

Et oui, "0.1 + 0.2" n'est pas égal à "0.3" ! Ces trois valeurs ne peuvent en fait pas être représentées de manière exacte par l'ordinateur et donc "0.1 + 0.2" n'est qu'une approximation du rationnel "3 / 10", tandis que "0.3" en est une autre approximation.

Deux approximations du même nombre ne sont pas forcément égales, elles sont simplement très proches, à une distance à peu près égale à 0.0000000000000000555 !

Conclusion

Vous aurez l'occasion d'en savoir plus sur ces histoires d'approximation dans un prochain cours mais le sujet est assez technique, aussi ne vous en préoccupez pas en détail pour le moment.

La seule chose à retenir c'est que les calculs avec des nombres décimaux ne sont pas *exacts* mais *approchés*, à cause de ces petites erreurs. Aussi on utilisera autant que possible des entiers plutôt que des nombres décimaux. En particulier il ne faut **jamais** faire de tests d'égalité ou d'inégalité sur des nombres décimaux. Quant aux inégalités entre nombres décimaux on évitera aussi et on préférera calculer avec des entiers et ne passer vers des nombres décimaux que le plus tard possible dans le programme.

CHARGER UN MODULE :

Dans Python, toutes les fonctions (en particulier les fonctions mathématiques) ne sont pas disponibles par défaut. Elles sont rangées dans ce qu'on appelle des *modules* (ou *bibliothèques*) qu'on peut voir comme des "boites à outils". Quand on veut utiliser un "outil" il faut d'abord "ouvrir la boite". Ainsi, pour accéder aux fonctions mathématiques de façon simple on utilise la commande suivante :

```
from math import *
```

Elle peut se traduire par "importer tout ce qui est dans le module `math`".

Cette commande est à placer tout en haut du fichier et permet ensuite d'utiliser dans votre programme toutes les fonctions ou constantes définies dans ce module. On dit qu'on a *importé* le module.

Par exemple, pour un calcul de racine carrée :

```
from math import *

print(sqrt(100))
```

```
10.0
```

5 – ARRONDIS :

Arrondir un nombre décimal, c'est le transformer en un entier. Il existe plusieurs manières de calculer des arrondis et nous en verrons deux dans ce cours :

- l'arrondi à l'entier **inférieur**,
- l'arrondi à l'entier **supérieur**.

Par exemple, si on a une température de 12.3 degrès alors les deux arrondis ci-dessus vont donner les valeurs 12 (entier inférieur) et 13 (entier supérieur).

En Python pour calculer des arrondis il faut tout d'abord avoir importé le module *math* :

```
from math import *
```

Pour calculer les arrondis on fait ainsi :

```
# Entier inférieur (partie entière)
arrondiInf = floor(12.3)
print(arrondiInf)

# Entier supérieur
arrondiSup = ceil(12.3)
print(arrondiSup)
```

```
12
13
```

Attention si le nombre est négatif !

```
# Entier inférieur (partie entière)
arrondiInf = floor(-12.3)
print(arrondiInf)

# Entier supérieur
arrondiSup = ceil(-12.3)
print(arrondiSup)
```

```
-13
-12
```

Selon la situation, ce sera à vous de déterminer de quelle fonction d'arrondi vous avez besoin.

Comment se souvenir des noms de ces fonctions ?

- `floor` comme "floor" la traduction en anglais du mot "sol" : le sol est en bas, donc on arrondi à l'entier inférieur
- `ceil` comme "ceiling" la traduction en anglais du mot "plafond" : le plafond est en haut, donc on arrondi à l'entier supérieur

Remarque : comme vous avez pu le voir, le résultat donné par ces fonctions est à chaque fois un entier et pas un nombre décimal représentant un entier. Dit autrement, on a bien "12" et pas "12.0" comme résultat.

FAIRE DES ARRONDIS AU PLUS PROCHE :

Dans un précédent cours nous avons vu comment calculer un arrondi **à l'entier inférieur** ou **à l'entier supérieur**. Nous allons maintenant voir comment calculer un arrondi **au plus proche** pour lequel, de manière naturelle, la valeur 1.3 sera arrondie à l'entier 1 et la valeur 1.8 sera arrondie à l'entier 2.

Il ne faut pas oublier d'importer la bibliothèque de maths (comme pour les fonctions `floor()` et `ceil()`), ensuite il suffit d'utiliser la fonction `round()` :

```
arrondiPro = round(12.3)
print(arrondiPro)
```

```
12
```

Pour se souvenir du nom de cette fonction, penser que "round" est la traduction en anglais du mot "arrondi".

Cas du 0.5 : différentes possibilités

Comment arrondir la valeur 1.5 ? Faut-il l'arrondir à 1 ou à 2 ? Il existe plusieurs choix possibles avec tous une bonne justification. Voici les principaux :

1. Au plus petit : si deux choix sont possibles, prendre le plus petit.
2. Au plus grand : si deux choix sont possibles, prendre le plus grand.
3. Vers zéro : si deux choix sont possibles, prendre le plus proche de zéro.
4. Vers l'infini : si deux choix sont possibles, prendre le plus éloigné de zéro.
5. Bancaire : si deux choix sont possibles, prendre celui qui est pair.

Cas du 0.5 : ce que fait Python

Alors, quelle est la technique utilisée en Python ? Regardons sur quelques valeurs !

```
print(round(-1.5))
print(round(-0.5))
print(round(0.5))
print(round(1.5))
```

```
-2
0
0
2
```

Il s'agit donc d'un arrondi "bancaire" quand il y a deux choix possibles.

6 – OPERATEURS DIVISION ENTIERE ET RESTE :

Vous avez 666 allumettes que vous voulez répartir par boites de 13, combien de boites pleines y aura-t-il ? S'il y a une boite non-pleine combien d'allumettes contiendra-t-telle ?

Ces questions sont liées à ce qu'on appelle la division euclidienne (ou division entière), c'est-à-dire qu'on souhaite trouver *nbBoites* et *nbReste* tels que :

- 666 = 13 * *nbBoites* + *nbReste*
- 0 <= *nbReste* < 13

La première condition signifie qu'on ne perd aucune allumette et la seconde que la boite non-pleine (si elle existe, c'est à dire si *nbReste* est différent de 0) ne peut pas contenir 13 allumettes (elle serait pleine sinon).

En Python il est possible de calculer *nbBoites* et *nbReste* très facilement, à l'aide de deux nouveaux opérateurs :

```
nbBoites = 666 // 13
nbReste = 666 % 13
print(nbBoites)
print(nbReste)
```

```
51
3
```

En terme de division euclidienne on a donc que, pour *a* et *b* deux entiers :

- `a // b` donne le *quotient* de la division euclidienne de *a* par *b*
- `a % b` donne le *reste* de la division euclidienne de *a* par *b*

Que se passe-t-il si au lieu de 666 on avait choisi un dividende négatif (on supposera que le diviseur est positif) ?

On ne peut plus parler d'allumettes mais imaginons qu'on doive payer 666 euros pour une anthologie de Heavy Métal (42 DVD !) et qu'on n'ait que des billets de 50 euros. On va donc donner 700 euros (soit 14 billets) et on va nous rendre 34 euros . Comme on paye quelque chose on a moins d'argent dans notre portefeuille, donc on perd de l'argent :

- -666 = 50 * (-14) + 34
- 0 <= 34 < 50

On a donc perdu 14 billets de 50 mais on a récupéré 34 euros.

Si on essaie d'écrire cela en Python, on a :

```
print(-666 // 50)
print(-666 % 50)
```

```
-14
34
```

La division euclidienne marche donc pour des nombres positifs et négatifs et garantit que si on a

```
quotient = dividende // diviseur
reste = dividende % diviseur
```

alors

- *dividende* = *diviseur* * *quotient* + *reste*
- 0 ≤ *reste* < *diviseur*

Un exemple fréquent d'utilisation de ces opérateurs :

```
nombre = int(input())
if (nombre % 2) == 0:
   print("Le nombre est pair")
```

En effet, si le reste vaut zéro c'est que *nombre* = *2 x quotient* donc *nombre* est divisible par 2.

Priorité des opérateurs division entière et reste :

Vous avez déjà appris que la priorité des opérateurs mathématiques (+, -, *, /) était la même en Python qu'en mathématiques. Ainsi on peut écrire de manière équivalente :

```
print(1 + 2 * 3 + 4 / 5)
print(1 + (2 * 3) + (4 / 5))
```

```
7.8
7.8
```

On commence donc par effectuer toutes les multiplications et divisions avant les additions et soustractions.

La priorité des opérateurs `//` et `%` est la même que celle des opérateurs de multiplication et division. Les calculs de quotient et reste se font donc avant les additions et soustractions. Les codes suivants sont donc équivalents :

```
print(10 + 20 // 3 + 42 % 5)
print(10 + (20 // 3) + (42 % 5))
```

```
18
18
```

Cela peut devenir un peu difficile sur des cas plus complexes. Ainsi quel serait à votre avis le résultat du calcul suivant ? Où faudrait-il mettre les parenthèses pour avoir le même résultat ?

```
print(100 * 200 // 300 // 40 % 50)
```

↳
```
1
```

Voici deux exemples de parenthèsages possibles, avec le résultat associé.

```
print(100 * (200 // (300 // (40 % 50))))
print((((100 * 200) // 300) // 40) % 50)
```

↳
```
2800
1
```

Il faut donc utiliser ce deuxième parenthèsage, c'est-à-dire effectuer le calcul de la gauche vers la droite (règle usuelle en mathématiques).

Vous avez pu voir sur cet exemple que sans parenthèses on peut rapidement se poser des questions et risquer de faire des erreurs (ne pas faire le bon calcul). On tâchera donc de ***toujours*** mettre des parenthèses quand le calcul peut sembler ambigu.