Types des expressions (Programmation 3)

I – Types d'expressions

1.1 - Encoder une information

Encoder : transformer une information réelle en suite de **0** et de **1** avec une technique connue.

Information → ENCODAGE → 010101011100...

Pourquoi doit-on connaître la technique utilisée pour encoder notre information ?

Simplement car lorsqu'on va vouloir lire l'information en mémoire, on va retomber sur notre suite de 0 et de 1. Si on ne sait pas la décoder, ca il ne va pas être facile de retrouver l'information cachée dans cette suite mystérieuse.

010101011100... → DECODAGE → Information

C'est dans la partie DONNEES de ce site que se trouvent les activités qui expliquent concrêtement comment encoder une information.

Ici, nous ne ferons qu'utiliser les techniques d'encodage préféfinies dans Python.

1.2 - Les integers

Les entiers sont encodés sous un type de données qu'on nomme les **integers**. En Python, on peut trouver le type utilisé pour gérer une donnée en utilisant la fonction native **type()**. Les integers sont nommés **int** dans le cadre du langage Python.

```
>>> type(5)
<class 'int'>
```

Voici les opérateurs classiques avec ces int :

- **addition** **+** : **int + int → int**
  Traduction : une addition entre un integer et un integer renvoie une valeur integer.
- **soustraction** **-** : **int - int → int**
  Traduction : une soustraction entre un integer et un integer renvoie une valeur integer.
- **multiplication** ***** : **int * int → int**
- **division** **/** : **int / int → float**
  float ? C'est le type utilisé pour les nombres à virgule.

  En Python, la **virgule** est représentée par un **point** (notation anglosaxonne).
- **quotient** **//** : **int // int → int**

  Tout simplement car on peut faire au maximum 4 tas de 5 avec 23.
- **reste** **%** : **int % int → int**

  Si on fait 4 tas de 5 avec 23, cela fait 20. Il reste bien 3 : 23 = 4*5 + 3.
- **puissance** ****** : **int ** int → int**

`>>> 4 + 5` `9`	`>>> 4 - 5` `-1`	`>>> 4 * 5` `20`
`>>> 23 / 5` `4.6`	`>>> 23 // 5` `4`	`>>> 23 % 5` `3`
`>>> 3 ** 2` `9`		

1.3 - Les floats

Les nombres à virgule sont encodés sous un type qu'on nomme les **floats**.

```
>>> type(5.0)
<class 'float'>
```

On retrouve tous les opérateurs classiques.

- **addition** avec **+** :

  float + float → float
  float + int → float
  int + float → float

- **soustraction** avec **-** :

  float - float → float
  float - int → float
  int - float → float

- **multiplication** * :

  float * float → float
  float * int → float
  int * float → float

- **division** avec **/** :

  float / float → float
  float / int → float
  int / float → float

- **quotient** d'une division avec `//`
  `float // float → float`
  `float // int → float`
  `int // float → float`
  On peut utiliser cet opérateur, il renvoie le bon résultat, mais en utilisant l'encodage du float.

- **reste** d'une division euclidienne avec %
  `float % float → float`
  `float % int → float`
  `int % float → float`
  Encore une fois, cela fonctionne mais... on obtiendra toujours un float.

- **puissance** avec **
  `float ** float → float`
  `int ** float → float`
  `float ** int → float`

`>>> 1.0 + 5` `6.0`	`>>> 5 - 4.5` `0.5`	`>>> 10 * 1.2` `12.0`
`>>> 23 / 10.0` `2.3`	`>>> 23 // 5.0` `4.0`	`>>> 23.0 % 5` `3.0`
`>>> 5.0 ** 2` `25.0`		

1.4 - Le problème des flottants

Tout nombre stocké sous forme d'**un integer est toujours exactement enregistré.**

`7 → ENCODAGE → DECODAGE → 7`

Certains nombres stockés sous forme d'un float sont enregistrés approximativement. On ne parvient donc pas à relire exactement ce qu'on a enregistré.

`0.1 → ENCODAGE → DECODAGE →  0.10000000000000001`

Ce n'est pas un problème lié à Python : c'est juste un problème lié aux nombres réels.

Conséquence : puisqu'on n'enregistre pas exactement le nombre mais une approximation du nombre, **les calculs réalisés à partir de l'information stockée seront approximatifs** aussi.

```
>>> 4.8 - 4.2          >>> 3 * 0.1
0.5999999999999996     0.30000000000000004
```

Le plus perturbant, c'est que cela fonctionne parfois :

```
>>> 0.75 - 0.25            >>> 3 * 0.25
0.5                        0.75
```

Conclusion temporaire :

1. Le type float encode certains nombres réels de façon approximative.
2. Pour réaliser des calculs exacts, on ne passe pas des flottants.

1.5 - Les strings

Le type permettant de gérer les textes se nomme souvent **string**. String est un mot anglais qui signifie "chaîne" en français. Exemple avec le mot bonjour :

`B → o → n → j → o → u → r`

En Python, les **strings** sont gérés par le type nommé **str**.

Comment déclarer un string ?

Une chaîne de caractères est délimitée par

1. un caractère indiquant le début
2. un caractère indiquant la fin.

Avec Python, on peut utiliser :

- deux guillemets simples : `'Bonjour'`,
- deux guillemets doubles : `"Bonjour"` ,
- mais jamais de mélange. Si on mélange, l'interpréteur Python perd le fil : c'est une **erreur de syntaxe**.

Quels opérateurs sur les strings ?

On va retrouver une partie des symboles de opérateurs applicables sur les valeurs numériques : la syntaxe sera la même mais la sémantique sera différente.

- **concaténation** avec `+` : `str + str → str`

```
>>> "bon" + "jour"      >>> '5' + '5'
'bonjour'               55'
```

```
>>> 'Bonj' + 5
TypeError: can only concatenate str
(not "int") to str
```

- **répétition** avec `*` :
  `str * int → str`
  `int * str → str`

```
>>> "Bon" * 2
'BonBon'
```

```
>>> 4 * "Pom ! "
'Pom ! Pom ! Pom ! Pom ! '
```

```
>>> '#' * 20
'####################'
```

- **et les autres ?**

  Les autres opérateurs (- / // % **">) ne sont pas implémentés sur les strings.

1.6 - Le type booléen

Booléens

Lorsqu'on utilise certains opérateurs, l'interpréteur Python est capable de comprendre qu'on lui pose une question. Il va alors évaluer votre expression et répondre à l'aide d'un booléen : la réponse ne peut prendre que deux valeurs :

- **`True`** si c'est vrai
- **`False`** sinon.

Pas de réponse de type "peut être". Néanmoins, la demande peut provoquer une erreur.

Opérateurs "strictement supérieur à" `>`

`int|float > int|float → bool`

Le signe | veut dire OU. On peut donc placer un entier ou un flottant.

"*40 est-il strictement supérieur à 30 ?*"

```
>>> 40 > 30
True
```

`str > str → bool`

"*Le mot "crocodile" est-il derrière le mot "tortue" dans le dictionnaire ?*"

```
>>> "crocodile" > "tortue"
False
```

On voit qu'on utlise ici l'ordre **lexicographique**, l'ordre du dictionnaire.

Opérateur d'égalité avec **`==`**

Puisque = est déjà utilisé pour l'affectation, les concepteurs de Python ont choisi la syntaxe d'un double signe égal pour le test d'égalité.

"*40 est-il identique à 3 * 10 ?*"

```
>>> 40 == 3 * 10
False
```

"*40 est-il identique à 4 * 10 ?*"

```
>>> 40 == 4 * 10
True
```

"*0.3 est-il identique à 3 * 0.1 ?*"

```
>>> 0.3 == 3 * 0.1
False
```

Opérateur de différence avec **`!=`**

Puisque ≠ n'est pas facilement accessible sur un clavier, les concepteurs de Python ont choisi la syntaxe `!=` pour le test de différence. Dans d'autres langages, c'est parfois <>.

"*40 est-il différent de 3 * 10 ?*"

```
>>> 40 != 3 * 10
True
```

"*40 est-il différent de 4 * 10 ?*"

```
>>> 40 != 4 * 10
False
```

Autres opérateurs de comparaison

"*40 est-il strictement inférieur à 30 ?*"

```
>>> 40 < 30
False
```

"*40 est-il inférieur ou égal à 30 ?*"

```
>>> 40 <= 30
False
```

"*40 est-il supérieur ou égal à 30 ?*"

```
>>> 40 >= 30
False
```

Et encore d'autres choses

Voici un dernier exemple avec le mot-clé **`in`** qui permet de savoir si un élément a est présent dans un élément b.

Si la réponse est True : a apparaît dans b.
Sinon, la réponse sera False.

*« Le mot **`"soja"`** apparaît-il dans la composition fournie ? »*

```
>>> "soja" in "Sucre ; huile de colza ; eau ;
chocolat en poudre 15 % (sucre, pâte de cacao)
; farine de blé ; poudre d'œuf ;
stabilisants : glycérol, gomme xanthane ;
cacao en poudre 2,5 % ; poudre à lever : E450,
E500 ; amidon de blé ; conservateur : E202 ;
sel. Traces de lait."
False
```

*Le mot **"blé"** apparaît-il dans la composition fournie" ?*

```
>>> "blé" in "Sucre ; huile de colza ; eau ;
chocolat en poudre 15 % (sucre, pâte de cacao)
; farine de blé ; poudre d'œuf ;
stabilisants : glycérol, gomme xanthane ;
cacao en poudre 2,5 % ; poudre à lever : E450,
E500 ; amidon de blé ; conservateur : E202 ;
sel. Traces de lait."
True
```

Comparaison et flottant

Jamais de programme comportant un test d'égalité entre flottants.

II – Exercices

III – Convertir

3.1 Créer un nouveau contenu "vide" mais du bon type

Création d'un contenu vide mais du bon type

- **int()** pour une donnée de type **int**

```
>>> i = int()
>>> i
0
```

- **float()** pour une donnée de type **float**

```
>>> f = float()
>>> f
0.0
```

- **str()** pour une donnée de type **str**

```
>>> s = str()
>>> s
''
```

- **bool()** pour une donnée de type **bool**

```
>>> b = bool()
>>> b
False
```

Déclaration directe

- pour une donnée de type **int**

```
>>> i = 0
>>> i
0
```

- pour une donnée de type **float**

```
>>> f = 0.0
>>> f
0.0
```

- pour une donnée de type **str**

```
>>> s = ""
>>> s
''
```

- pour une donnée de type bool

```
>>> b = False
>>> b
False
```

3.2 Tenter de créer un integer avec `int()` : Cela ne fonctionne pas toujours : voir le site pour info.

3.3 Tenter de créer un float avec `float()` : Cela ne fonctionne pas toujours : voir le site pour info.

3.4 Tenter de créer un string avec `str()` : Pas d'erreur particulière à noter.

3.5 Tenter de créer un booléen avec `bool()` : Pas d'erreur particulière, mais c'est à connaitre

La fonction fonctionne toujours :

- Si on envoie une **valeur nulle ou vide**, c'est **`False`**.
- Dans tous les autres cas, c'est **`True`**.

```
>>> b = bool(0)
>>> b
False

>>> b = bool(5)
>>> b
True

>>> b = bool(5.2)
>>> b
True

>>> b = bool(0.0)
>>> b
False

>>> b = bool("5")
>>> b
True

>>> b = bool("5.2")
>>> b
True

>>> b = bool("bonjour")
>>> b
True

>>> b = bool("")
>>> b
False

>>> b = bool(False)
>>> b
False

>>> b = bool(True)
>>> b
True
```