Types de base en Python

Entiers, flottants, chaînes et booléens

En Python, chaque valeur possède un **type** qui détermine ce qu'on peut en faire. On ne multiplie pas un mot par un autre mot, et on ne met pas un nombre en majuscules. Le type d'une donnée définit les opérations autorisées sur cette donnée.

Python propose quatre types de base, que l'on utilise au quotidien :

Type	Nom Python	Exemple
Entier	`int`	`42`
Flottant	`float`	`3.14`
Chaîne de caractères	`str`	`'Bonjour'`
Booléen	`bool`	`True`

① La fonction `type()`

Pour connaître le type d'une valeur ou d'une variable, on utilise la fonction `type()` :

```
print(type(42))          # <class 'int'>
print(type(3.14))        # <class 'float'>
print(type('Bonjour'))   # <class 'str'>
print(type(True))        # <class 'bool'>
```

C'est un outil précieux pour comprendre le comportement d'un programme, notamment lorsqu'une erreur survient.

Question.
Quel est le type de chacune des valeurs suivantes : `7`, `7.0`, `"7"`, `True` et `"True"` ?

Réponse.

Valeur	Type	Pourquoi
`7`	`int`	un nombre sans point décimal
`7.0`	`float`	le point décimal suffit à en faire un flottant
`"7"`	`str`	les guillemets en font une chaîne de caractères
`True`	`bool`	une des deux valeurs booléennes, écrite sans guillemets
`"True"`	`str`	entre guillemets, ce n'est plus un booléen mais une chaîne

On le vérifie avec la fonction `type()` :

```
print(type(7), type(7.0), type("7"), type(True), type("True"))
```

```
<class 'int'> <class 'float'> <class 'str'> <class 'bool'> <class 'str'>
```

En Python, c'est la **donnée** qui est typée, pas la variable. Une même variable peut donc référencer successivement des valeurs de types différents :

```
x = 42          # x référence un int
x = "bonjour"   # maintenant x référence un str
```

② Les entiers (`int`)

Un **entier** est un nombre sans partie décimale : positif, négatif ou nul.

```
age = 17
temperature = -5
population = 8000000000
print(type(age))  # <class 'int'>
```

Particularité de Python : contrairement à beaucoup de langages, la taille des entiers est **illimitée**. On peut manipuler des nombres aussi grands qu'on le souhaite sans risquer de dépassement :

```
grand_nombre = 2 ** 100
print(grand_nombre)  # 1267650600228229401496703205376
```

Division `/` et division entière `//`

Parmi les opérations sur les entiers (détaillées dans le cours sur les opérateurs), la division mérite une attention particulière. Voici deux divisions de 10 par 2, l'une avec une barre, l'autre avec deux barres :

```
print(10 / 2)
print(10 // 2)
```

Question.
Quels nombres ces deux lignes affichent-elles ?

Réponse.

```
5.0
5
```

La division `/` renvoie **toujours** un flottant, même quand le résultat tombe juste : `5.0`. La division `//` est la **division entière**, appelée aussi division euclidienne : elle donne le quotient, un entier, `5`.

```
print(type(10 / 2))   # <class 'float'>
print(type(10 // 2))  # <class 'int'>
print(7 // 2)         # 3 : le quotient de 7 par 2
```

Pour des nombres positifs, la division entière revient à supprimer la partie décimale du quotient : `7 / 2` vaut `3.5`, et `7 // 2` vaut `3`.

Exercice 1 *Entiers en Python*

1. Quel est le type de `2 ** 200` ? Vérifiez en Python.

2. Quel est le type de `100 / 10` ? Et de `100 // 10` ? Pourquoi cette différence ?

3. Un programme calcule le nombre de secondes dans un an : `365 * 24 * 60 * 60`. Quel est le résultat ? Quel est son type ?

③ Les flottants (`float`)

Un **flottant** (nombre à virgule flottante) est un nombre avec une partie décimale. En Python, on utilise le **point** comme séparateur décimal (convention anglo-saxonne) :

```
pi = 3.14159
taille = 1.75
negatif = -0.5
print(type(pi))  # <class 'float'>
```

Remarque : écrire `3.0` crée un `float`, pas un `int`. La seule présence du point décimal suffit à changer le type.

Calcul qui mêle les deux types : si un calcul mêle un `int` et un `float`, le résultat est un `float`. Par exemple, `3 + 4.0` vaut `7.0` et `2 * 1.5` vaut `3.0`.

Précision limitée

Les flottants sont stockés en mémoire selon la norme IEEE 754 sur 64 bits. Leur précision est limitée à environ quinze chiffres significatifs, ce qui entraîne parfois des résultats surprenants.

```
print(0.1 + 0.2)
```

Question.
Qu'affiche cette ligne ?

Réponse.

```
0.30000000000000004
```

Le résultat n'est pas exactement `0.3`. Ce n'est **pas** une erreur de Python : l'ordinateur écrit les nombres en binaire, avec un nombre limité de chiffres. Or `0.1` ne s'écrit pas exactement en binaire, de même que 1/3 ne s'écrit pas exactement en écriture décimale (0,333... à l'infini). L'ordinateur n'en garde qu'une partie : le nombre stocké est très proche de 0,1, mais pas exactement égal, d'où la petite erreur du résultat.

Conséquence pratique : on ne compare jamais deux flottants avec `==`. On vérifie plutôt que leur différence est très petite :

```
a = 0.1 + 0.2
b = 0.3
print(a == b)              # False (piège !)
print(abs(a - b) < 1e-10)  # True (bonne méthode)
```

La fonction `abs(x)` renvoie la valeur absolue de `x`, c'est-à-dire sa distance à zéro : `abs(-3)` vaut `3`. L'écriture `1e-10` est la notation scientifique de 10□¹□, soit 0,0000000001 ; c'est un `float`.

Exercice 2 *Pièges des flottants*

1. Calculez `0.1 + 0.1 + 0.1` en Python. Le résultat est-il exactement `0.3` ? Vérifiez avec `==`.

2. Calculez `0.1 + 0.1 + 0.1 - 0.3`. Le résultat est-il exactement `0` ?

3. Proposez une expression qui teste si deux flottants `a` et `b` sont « presque égaux » à 10^{-9} près.

④ Les chaînes de caractères (`str`)

Une **chaîne de caractères** est une séquence de caractères entourée de guillemets simples ou doubles :

```
prenom = 'Alice'
message = "Bonjour le monde"
print(type(prenom))  # <class 'str'>
```

Les deux notations sont équivalentes. Les guillemets doubles sont pratiques lorsque la chaîne contient une apostrophe :

```
phrase = "L'informatique est passionnante"
```

Opérations de base sur les chaînes

```
# Longueur
mot = 'Python'
print(len(mot))      # 6

# Concaténation (assemblage)
prenom = 'Ada'
nom = 'Lovelace'
complet = prenom + ' ' + nom
print(complet)       # Ada Lovelace

# Répétition
print('ha' * 3)      # hahaha
```

Un nombre ou une chaîne ?

Voici le piège le plus fréquent des débutants. Ces deux lignes se ressemblent beaucoup :

```
print(42 + 1)
print("42" + "1")
```

Question.
Affichent-elles la même chose ?

Réponse.

```
43
421
```

Avec des nombres, le signe `+` calcule une somme : `43`. Entre guillemets, `"42"` et `"1"` sont des chaînes de caractères : le même signe `+` colle alors les deux chaînes l'une après l'autre, et on obtient la chaîne `"421"`.

À retenir. Les guillemets changent tout. Sans guillemets, `42` est un nombre ; avec des guillemets, `"42"` est une chaîne de caractères.

Attention. on ne peut **pas** additionner une chaîne et un nombre directement.

```
age = 17
print("J'ai " + age + " ans")
```

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

Il faut convertir le nombre en chaîne avec `str()`, ou utiliser une f-string, plus simple :

```
age = 17
print("J'ai " + str(age) + " ans")
print(f"J'ai {age} ans")
```

```
J'ai 17 ans
J'ai 17 ans
```

Les chaînes de caractères seront étudiées en détail dans un chapitre dédié.

Exercice 3 *Chaînes de caractères*

1. Quelle est la longueur de `"Bonjour"` (nombre de caractères) ? Et de `""` (chaîne vide) ?

2. Que produit `"3" + "4"` ? Et `3 + 4` ? Expliquez la différence.

3. Que produit `"ha" * 3` ? Et `3 * "ha"` ? Et `"ha" * 0` ?

5 Les booléens (`bool`)

Un **booléen** ne peut prendre que deux valeurs : `True` (vrai) ou `False` (faux). Ces valeurs commencent obligatoirement par une **majuscule**.

```
majeur = True
inscrit = False
print(type(majeur))  # <class 'bool'>
```

Les booléens apparaissent naturellement comme résultat des comparaisons (étudiées dans le cours sur les opérateurs) :

```
print(5 > 3)    # True
print(2 == 7)   # False
```

La valeur spéciale `None`

Python dispose d'une valeur particulière, `None`, qui représente l'**absence de valeur**. Son type est `NoneType` :

```python
resultat = None
print(type(resultat))  # <class 'NoneType'>
```

On utilise `None` pour initialiser une variable dont on ne connaît pas encore la valeur. On le rencontre aussi comme valeur de retour de certaines fonctions (ce point sera détaillé dans le chapitre sur les fonctions). Pour savoir si une variable vaut `None`, on peut l'afficher : si `x` vaut `None`, `print(x)` affiche `None`.

Valeurs « vraies » et « fausses »

En Python, certaines valeurs sont considérées comme « fausses » dans un contexte booléen : `0`, `0.0`, `''` (la chaîne vide) et `None`. Toute autre valeur est considérée comme « vraie ». On peut le vérifier avec `bool()` ; cette règle servira surtout dans les conditions (chapitre 04).

```python
print(bool(0))      # False
print(bool(42))     # True
print(bool(''))     # False
print(bool('abc'))  # True
print(bool(None))   # False
```

Piège fréquent : la chaîne `"False"` est **vraie**, car c'est une chaîne non vide (cinq caractères). Seule la chaîne vide `""` est fausse.

Exercice 4 *Booléens et valeurs de vérité*

1. Quel est le type de `True` ? Et de `"True"` ?

2. Parmi les valeurs suivantes, lesquelles sont « fausses » au sens de `bool()` : `0`, `1`, `-1`, `0.0`, `""`, `"0"`, `None`, `False` ?

3. Que donne `bool("False")` ? Pourquoi ?

6 Conversions de types

Il est souvent nécessaire de convertir une valeur d'un type à un autre. Python offre des fonctions portant le nom du type cible :

```python
# Entier vers flottant
print(float(42))      # 42.0

# Chaîne vers entier
print(int('15'))      # 15

```

```
# Chaîne vers flottant
print(float('3.14'))  # 3.14

# Nombre vers chaîne
print(str(42))        # '42'
print(str(3.14))      # '3.14'
```

Dans l'autre sens, la conversion d'un flottant en entier réserve une surprise :

```
print(int(3.7))
print(int(-2.9))
```

Question.
Que donnent ces deux conversions ?

Réponse.

```
3
-2
```

La fonction `int()` **tronque** : elle supprime la partie décimale, sans arrondir. `int(3.7)` donne donc `3`, et non pas `4`, et `int(-2.9)` donne `-2`.

Pour arrondir, on utilise `round()` : `round(3.7)` donne `4`. Avec un second argument, `round(x, n)` arrondit `x` à `n` chiffres après la virgule : `round(3.14159, 2)` donne `3.14`.

La fonction `input()` renvoie une chaîne

La fonction `input()` demande à l'utilisateur de taper une valeur au clavier. Supposons qu'il tape `17`, et qu'on veuille calculer son âge l'année prochaine :

```
age = input("Ton âge ? ")   # l'utilisateur tape 17
print(age + 1)
```

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

Python affiche une erreur, car `input()` renvoie **toujours** une chaîne de caractères : `age` contient la chaîne `"17"`, et Python refuse d'ajouter un nombre à une chaîne. La solution consiste à convertir la saisie en entier avec `int()`, dès la saisie :

```
age = int(input("Ton âge ? "))   # l'utilisateur tape 17
print(age + 1)
```

```
18
```

Exemple pas à pas : saisie et conversion

```
age = input("Ton âge ? ")    # l'utilisateur tape 17
futur = int(age) + 10
print(f"Dans 10 ans : {futur}")
```

```
Dans 10 ans : 27
```

On suit le type de chaque valeur, étape par étape :

Étape	Instruction ou expression	Valeur	Type
1	`age = input("Ton âge ? ")`	`'17'`	`str`
2	`int(age)`	17	`int`
3	`futur = int(age) + 10`	27	`int`
4	`f"Dans 10 ans : {futur}"`	`'Dans 10 ans : 27'`	`str`

Sans la conversion `int(age)`, Python tenterait `'17' + 10` et produirait un `TypeError`, car on ne peut pas additionner une chaîne et un entier.

Conversions impossibles

Certaines conversions échouent et produisent une erreur `ValueError` :

```
# int('abc')    # ValueError : la chaîne ne représente pas un nombre
# float('abc')  # ValueError
```

```
print(int("3.14"))
```

Question.
Que donne `int("3.14")` ?

Réponse.

```
ValueError: invalid literal for int() with base 10: '3.14'
```

Une erreur : `int()` n'accepte que les chaînes qui représentent un nombre entier, et `"3.14"` contient un point. Pour convertir `'3.14'` en entier, il faut passer par un flottant intermédiaire : `int(float('3.14'))` donne `3`.

Exercice 5 *Conversions de types*

1. Sans exécuter, prédisez le résultat et le type de chaque expression :

   a) `int(4.9)`

b) `float(3)`

c) `str(42) + str(8)`

d) `int("12") + int("8")`

e) `bool("False")`

2. Écrivez un programme qui demande une température en degrés Celsius et affiche son équivalent en degrés Fahrenheit, arrondi à un chiffre après la virgule. Formule : $F = C \times 1{,}8 + 32$.

Exercice 6 *Synthèse — Types et erreurs* Pour chaque expression, indiquez si elle produit un `int`, un `float`, un `str`, un `bool`, ou une **erreur** :

1. `3 + 4`
2. `3 + 4.0`
3. `"3" + "4"`
4. `3 + "4"`
5. `10 / 5`
6. `int("12") * 2`
7. `float("abc")`

⑦ À RETENIR

1. Chaque valeur a un type : `int`, `float`, `str` ou `bool`. La fonction `type()` le donne.

2. Les guillemets font la chaîne : `42` est un entier, `"42"` une chaîne de caractères, et `"42" + "1"` vaut `"421"`.
3. Le point décimal fait le flottant : `7.0` est un `float`.
4. Les entiers ont une taille illimitée. La division `/` renvoie toujours un flottant, la division entière `//` un entier.
5. Les flottants sont approchés : on ne compare jamais deux flottants avec `==`.
6. `input()` renvoie toujours une chaîne : on la convertit avec `int()` ou `float()` avant de calculer.
7. `int()` supprime la partie décimale sans arrondir ; `round()` arrondit.
8. `True` et `False` s'écrivent avec une majuscule ; une chaîne non vide est « vraie », même `"False"`.

⑧ Résumé

Type	Nom Python	Exemples	Fonction de conversion
Entier	`int`	`42`, `-7`, `0`	`int()`
Flottant	`float`	`3.14`, `-0.5`	`float()`
Chaîne	`str`	`'Bonjour'`, `"42"`	`str()`
Booléen	`bool`	`True`, `False`	`bool()`