Booléens et alternative

Doc. 2,3

Objectif

Manipuler des expressions booléennes dans la console Python pour découvrir comment combiner des conditions, puis écrire des fonctions dont l'exécution dépend de ces conditions.

§1. Comment exprimer une condition ?

Q1. Tester dans la console :

```Python
>>> 3 > 2
>>> 3 == 2
>>> type(3 > 2)
>>> True
>>> False
>>> type(True)
```

Quel est ce nouveau type ? Combien de valeurs différentes peut-il prendre ?

Réponse : Le type `bool` (**booléen**) ne peut prendre que deux valeurs : `True` (vrai) ou `False` (faux). Une comparaison comme `3 > 2` est une **expression booléenne** : son évaluation produit un `bool`.

Q2. Une confusion fréquente oppose `=` et `==`. Prévoir le résultat de l'évaluation de chacune des instructions suivantes, puis tester dans la console :

```Python
>>> a = 5
>>> a == 5
>>> a
>>> a = (a == 5)
>>> a
```

Quelle instruction **modifie** la variable `a` ? Laquelle se contente de **comparer** sans rien modifier ?

Réponse : `a = 5` et `a = (a == 5)` sont des **affectations** (opérateur `=`) : elles modifient `a`. `a == 5` est une **comparaison** (opérateur `==`) : elle renvoie `True` sans modifier `a`. À la fin, `a` vaut `True`, un booléen.

Q3. Par analogie avec `>` et `==`, prévoir le résultat de l'évaluation de chacune des expressions suivantes, puis tester dans la console :

```Python
>>> a, b = 3, 5
>>> a < b
>>> a >= b
>>> a <= b
```

```
>>> a != b
>>> a + 2 == b
```

Pour chaque opérateur utilisé, indiquer en une phrase ce qu'il teste.

Réponse :

Opérateur	Teste si...
`<`	strictement inférieur
`>`	strictement supérieur
`<=`	inférieur ou égal
`>=`	supérieur ou égal
`==`	égal
`!=`	différent

On obtient successivement `True`, `False`, `True`, `True`, `True`.

Q4. Les opérateurs de comparaison s'utilisent aussi sur des chaînes de caractères, comparées dans l'ordre alphabétique. Prévoir le résultat de l'évaluation de chacune des expressions suivantes, puis tester dans la console :

```
>>> lettre, car = "i", "j"
>>> lettre <= car
>>> lettre == "w"
```
Python

Réponse : On obtient `True` puis `False` : `"i"` précède `"j"` dans l'ordre alphabétique.

Q5. Python peut convertir n'importe quelle valeur en booléen avec la fonction `bool`. Prévoir le résultat de l'évaluation de chacune des expressions suivantes, puis tester dans la console :

```
>>> bool(0)
>>> bool(1)
>>> bool(-5)
>>> bool("")
>>> bool("a")
>>> bool([])
```
Python

Quelle règle générale peut-on énoncer ?

Réponse : On obtient `False`, `True`, `True`, `False`, `True`, `False`. Une valeur **nulle ou vide** (`0`, `""`, `[]`, ...) est convertie en `False` ; toute autre valeur est convertie en `True`.

À retenir

Une **grandeur booléenne** ne prend que deux valeurs : `True` ou `False`. Une **expression booléenne** (comparaison, appel de `bool`, ...) est une expression dont l'évaluation produit une grandeur booléenne.

Ne pas confondre `=` (affectation) et `==` (comparaison).

Q6. **Sans utiliser Python**, indiquer si chacune des expressions suivantes vaut `True` ou `False`, en justifiant :

```
>>> 7 <= 7
>>> "abc" == "abd"
>>> 3 != 3.0
>>> "Ana" <= "ana"
```
Python

Vérifier ensuite dans la console.

Réponse :
- `7 <= 7` vaut `True` (l'égalité est incluse dans `<=`).
- `"abc" == "abd"` vaut `False` (le troisième caractère diffère).
- `3 != 3.0` vaut `False` : Python considère `3` et `3.0` comme des valeurs **numériquement égales**, même si leurs types diffèrent.
- `"Ana" <= "ana"` vaut `True` : dans l'ordre utilisé par Python, une majuscule est **toujours** considérée comme inférieure à une minuscule.

§2. Comment combiner plusieurs conditions ?

Q7. Prévoir le résultat de l'évaluation de chacune des expressions suivantes, puis tester dans la console :

```
>>> not True
>>> not False
```
Python

Que fait l'opérateur `not` ?

Réponse : `not` inverse une valeur booléenne : `not True` vaut `False`, `not False` vaut `True`.

Q8. Tester dans la console les quatre combinaisons possibles de l'opérateur `and`, puis compléter le tableau ci-dessous.

```
>>> True and True
>>> True and False
>>> False and True
>>> False and False
```
Python

A	B	A and B
True	True	
True	False	
False	True	
False	False	

Réponse :

A	B	A and B

True	True	True
True	False	False
False	True	False
False	False	False

`A and B` ne vaut `True` que si `A` **et** `B` valent `True`.

Q9. De la même façon, tester les quatre combinaisons de l'opérateur `or`, puis compléter le tableau ci-dessous.

```
>>> True or True
>>> True or False
>>> False or True
>>> False or False
```

A	B	A or B
True	True	
True	False	
False	True	
False	False	

Réponse :

A	B	A or B
True	True	True
True	False	True
False	True	True
False	False	False

`A or B` ne vaut `False` que si `A` **et** `B` valent `False`.

Q10. **Essentiel pour l'épreuve écrite.** Tester dans la console, une instruction à la fois :

```
>>> False and (1 / 0)
>>> True or (1 / 0)
>>> True and (1 / 0)
>>> (1 / 0) and True
```

L'une de ces quatre instructions ne provoque **pas** d'erreur alors qu'elle contient une division par zéro. Laquelle ? Pour chaque instruction, indiquer si Python a eu besoin d'évaluer `1 / 0` pour connaître le résultat.

Réponse :

- `False and (1 / 0)` : Python sait déjà que le résultat est `False` dès qu'il lit `False` ; il **n'évalue pas** `1 / 0`. Aucune erreur.
- `True or (1 / 0)` : Python sait déjà que le résultat est `True` dès qu'il lit `True` ; il **n'évalue pas** `1 / 0`. Aucune erreur.

— `True and (1 / 0)` : Python doit évaluer le second opérande pour connaître le résultat ; `ZeroDivisionError`.
— `(1 / 0) and True` : Python évalue toujours le **premier** opérande en premier, quoi qu'il arrive ; `ZeroDivisionError` immédiate.

Évaluation en court-circuit

Python évalue toujours une expression de gauche à droite, mais s'arrête dès que le résultat est déterminé, sans évaluer le reste :

— avec `and`, si le premier opérande vaut `False`, le second n'est pas évalué (le résultat est déjà `False`) ;
— avec `or`, si le premier opérande vaut `True`, le second n'est pas évalué (le résultat est déjà `True`).

Q11. **Sans utiliser Python.** On donne `e = False`, `f = False` ; les valeurs de `d` ne sont pas connues. Déterminer, en justifiant par l'évaluation en court-circuit, la valeur de l'expression suivante (et si `d` doit être connue pour y parvenir) :

```
>>> e or ((not e) and f and d)
```
Python

Vérifier ensuite dans la console avec `d = 1 / 0` (qui provoquerait une erreur si Python l'évaluait).

Réponse : `e` vaut `False`, donc le `or` doit évaluer le second opérande : `(not e)` vaut `True`, puis `True and f` vaut `False` (car `f` vaut `False`). Le `and` s'arrête alors : `d` n'est **jamais évalué**. L'expression complète vaut `False`, sans erreur même si `d = 1 / 0`.

§3. Comment exécuter des instructions sous condition ?

Q12. Dans la partie principale de l'éditeur, écrire puis exécuter :

```
x = -5
if x < 0:
    print("x est négatif")
print("fin du programme")
```
Python

Relever ce qui s'affiche. Remplacer `-5` par `5`, exécuter à nouveau et relever ce qui s'affiche. Quelle instruction n'est exécutée que dans certains cas ?

Réponse : Avec `x = -5`, les deux lignes s'affichent. Avec `x = 5`, seule `"fin du programme"` s'affiche. L'instruction `print("x est négatif")`, indentée sous le `if`, n'est exécutée que lorsque `x < 0` est `True`.

Alternative simple

`if <expression booléenne>:` suivi d'un bloc **indenté** ne fait exécuter ce bloc que si l'expression booléenne vaut `True`. Si elle vaut `False`, le bloc est ignoré et le flot d'exécution se poursuit normalement après.

Q13. Modifier le programme précédent pour qu'il affiche `"x est négatif"` si `x < 0`, et `"x est positif ou nul"` sinon. Tester avec `x = -5` puis `x = 5`.

Réponse :

```python
x = -5
if x < 0:
    print("x est négatif")
else:
    print("x est positif ou nul")
print("fin du programme")
```

À retenir

`if <exprBool>:`, suivi d'un bloc indenté, n'exécute ce bloc que si `<exprBool>` est `True` :

```python
if <exprBool>:
    instructions
```

Avec `else`, on exécute le premier bloc si `<exprBool>` est `True`, le second sinon :

```python
if <exprBool>:
    ...
else:
    ...
```

Q14. On veut comparer deux nombres `i` et `j` et afficher le plus grand, ou `"égaux"` s'ils sont identiques. Dans la partie principale de l'éditeur, écrire puis exécuter, pour plusieurs valeurs de `i` et `j` (par exemple `3, 7` puis `7, 3` puis `4, 4`):

```python
i, j = 3, 7
if i < j:
    message = "j"
elif i > j:
    message = "i"
else:
    message = "égaux"
print(message)
```

Combien de blocs peuvent être exécutés au maximum pour une même exécution du programme ?

Réponse : Un seul bloc est exécuté : dès que Python trouve la première condition vraie (`i < j` ou `i > j`), il exécute le bloc correspondant et ignore les suivants. Si aucune des conditions n'est vraie, c'est le bloc `else` qui s'exécute.

Q15. Réécrire le programme précédent **sans utiliser** `elif`, à l'aide de deux `if`/`else` imbriqués. Vérifier que les deux versions produisent le même résultat pour `i, j = 3, 7`, `7, 3` et `4, 4`.

Réponse :

```python
i, j = 3, 7
```

```python
if i < j:
    message = "j"
else:
    if i > j:
        message = "i"
    else:
        message = "égaux"
print(message)
```

elif

Une chaîne `if` / `elif` / … / `else` est équivalente à des `if` / `else` imbriqués, mais sa syntaxe est bien plus lisible dès que le nombre de cas augmente. La branche `else` finale n'est pas obligatoire.

Q16. **Sans utiliser Python.** En reprenant le programme de comparaison de `i` et `j`, déterminer la valeur de `message` pour chacun des couples suivants, en indiquant à chaque fois quelle condition a été la première trouvée vraie :

- `i, j = 2, 9`
- `i, j = 9, 2`
- `i, j = -4, -4`

Vérifier ensuite dans la console ou dans la partie principale de l'éditeur.

Réponse :

- `i, j = 2, 9` : `i < j` est vraie en premier → `message = "j"`.
- `i, j = 9, 2` : `i < j` est fausse, `i > j` est vraie → `message = "i"`.
- `i, j = -4, -4` : `i < j` et `i > j` sont fausses toutes les deux → c'est le `else` qui s'exécute, `message = "égaux"`.

§4. Qu'est-ce qu'un prédicat ?

Q17. Écrire, dans la partie principale de l'éditeur, la fonction `est_positif` dont la spécification est :

```python
def est_positif(x: int | float) -> bool:
    """Retourne True si x est positif ou nul, False sinon."""
```

Tester ensuite `est_positif(3)` et `est_positif(-3)` dans la console. Que remarquez-vous sur le type de la valeur **retournée** par cette fonction ?

Réponse :

```python
def est_positif(x: int | float) -> bool:
    """Retourne True si x est positif ou nul, False sinon."""
    return x >= 0
```

La fonction retourne directement le résultat d'une expression booléenne : son type de retour est `bool`.

Prédicat

On appelle **prédicat** une fonction qui retourne un booléen. Par convention, le nom d'un prédicat commence par `est_`.

Q18. Tester dans la console :

```
>>> 1 == 1
>>> 1 / 49 * 49 == 1
```

L'une des deux instructions ne donne pas le résultat attendu mathématiquement. Laquelle, et pourquoi (on pourra tester `1 / 49 * 49` seule) ?

Réponse : `1 / 49 * 49 == 1` vaut `False` : le calcul `1 / 49 * 49` donne une valeur très proche de `1` mais pas exactement égale, à cause des arrondis inévitables du calcul flottant. Comparer deux `float` avec `==` est donc risqué.

À retenir

On ne compare **jamais** deux `float` avec `==` : les arrondis du calcul flottant peuvent rendre deux valeurs mathématiquement égales différentes en mémoire, et l'égalité échoue. On vérifie plutôt que leur écart est **suffisamment petit** (voir `est_egal` ci-dessous).

Q19. Écrire le prédicat `est_egal` dont la spécification est :

```
def est_egal(x: int | float, y: int | float) -> bool:
    """Retourne True si abs(x - y) < epsilon, où epsilon = 1e-5.

    >>> est_egal(1 / 49 * 49, 1)
    True
    """
```

Tester ensuite son appel dans la console.

Réponse :

```
def est_egal(x: int | float, y: int | float) -> bool:
    """Retourne True si abs(x - y) < epsilon, où epsilon = 1e-5.

    >>> est_egal(1 / 49 * 49, 1)
    True
    """
    epsilon = 1e-5
    return abs(x - y) < epsilon
```

Q20. Écrire le prédicat `est_pair` dont la spécification est :

```
def est_pair(n: int) -> bool:
    """Retourne True si n est un nombre pair, False sinon."""
```

Réponse :

```python
def est_pair(n: int) -> bool:
    """Retourne True si n est un nombre pair, False sinon."""
    return n % 2 == 0
```

Q21. Écrire le prédicat `est_entier` dont la spécification est :

```python
def est_entier(n: int) -> bool:
    """Retourne True si n ne possède pas de partie décimale, False sinon."""
```

Consulter `help(int)` dans la console : comment la fonction `int` peut-elle aider à écrire ce prédicat ?

Réponse :

```python
def est_entier(n: int) -> bool:
    """Retourne True si n ne possède pas de partie décimale, False sinon."""
    return n == int(n)
```

`int(n)` tronque la partie décimale de `n` ; si le résultat est égal à `n`, c'est que `n` n'en avait pas.

Q22. Écrire le prédicat `est_naturel` dont la spécification est :

```python
def est_naturel(n: int) -> bool:
    """Détermine si n est un entier naturel (entier positif ou nul).

    >>> est_naturel(5)
    True
    >>> est_naturel(-5)
    False
    >>> est_naturel(2.3)
    False
    """
```

Cette fonction doit utiliser les fonctions `est_positif` et `est_entier` des exercices précédents, combinées avec `and`.

Réponse :

```python
def est_naturel(n: int) -> bool:
    """Détermine si n est un entier naturel (entier positif ou nul).

    >>> est_naturel(5)
    True
    >>> est_naturel(-5)
    False
    >>> est_naturel(2.3)
```

```
    False
    """
    return est_positif(n) and est_entier(n)
```

Q23. **Sans utiliser Python.** En réutilisant la définition `est_naturel(n) = est_positif(n) and est_entier(n)`, déterminer à la main le résultat de `est_naturel(-3.5)`, `est_naturel(4.0)` et `est_naturel(7)`. Pour le premier cas, indiquer si `est_entier` a besoin d'être évaluée.

Vérifier ensuite dans la console.

Réponse :

— `est_naturel(-3.5)` : `est_positif(-3.5)` vaut déjà `False` ; grâce à l'évaluation en court-circuit du `and`, `est_entier(-3.5)` n'est **pas évaluée**. Résultat : `False`.

— `est_naturel(4.0)` : `est_positif(4.0)` vaut `True`, donc `est_entier(4.0)` est évaluée et vaut aussi `True`. Résultat : `True`.

— `est_naturel(7)` : les deux prédicats valent `True`. Résultat : `True`.

Q24. Tester dans la console `est_naturel("coucou")`. Que se passe-t-il ? Écrire le prédicat `est_naturel_2`, qui reprend `est_naturel` mais vérifie d'abord, à l'aide d'une instruction `assert` et de la fonction `isinstance`, que l'argument est bien un nombre :

```python
def est_naturel_2(n: int) -> bool:
    """Détermine si n est un entier naturel.
    Erreur si n n'est pas un nombre.

    >>> est_naturel_2(5)
    True
    >>> est_naturel_2(-5)
    False
    >>> est_naturel_2(2.3)
    False
    >>> est_naturel_2("coucou")
    Traceback ...
    AssertionError: n doit être un nombre
    """
```

Réponse : `est_naturel("coucou")` provoque une `TypeError` **peu explicite**, quelque part dans le calcul, sans indiquer clairement quel argument est en cause.

```python
def est_naturel_2(n: int) -> bool:
    """Détermine si n est un entier naturel.
    Erreur si n n'est pas un nombre.

    >>> est_naturel_2(5)
    True
```

```
    >>> est_naturel_2(-5)
    False
    >>> est_naturel_2(2.3)
    False
    >>> est_naturel_2("coucou")
    Traceback ...
    AssertionError: n doit être un nombre
    """
    assert isinstance(n, int) or isinstance(n, float), "n doit être un nombre"
    return est_naturel(n)
```

Q25. Écrire le prédicat `est_plus_grand` dont la spécification est :

```python
def est_plus_grand(x: int | float, y: int | float) -> bool:
    """Retourne True si x > y, False sinon."""
```

Écrire ensuite la fonction `le_plus_grand`, qui doit utiliser `est_plus_grand` :

```python
def le_plus_grand(x: int | float, y: int | float) -> int | float:
    """Retourne x si x > y, y sinon."""
```

Réponse :

```python
def est_plus_grand(x: int | float, y: int | float) -> bool:
    """Retourne True si x > y, False sinon."""
    return x > y

def le_plus_grand(x: int | float, y: int | float) -> int | float:
    """Retourne x si x > y, y sinon."""
    if est_plus_grand(x, y):
        return x
    else:
        return y
```

Q26. Écrire la fonction `valeur_absolue`, qui doit utiliser le prédicat `est_positif` :

```python
def valeur_absolue(x: int | float) -> int | float:
    """Retourne la valeur absolue de x."""
```

Réponse :

```python
def valeur_absolue(x: int | float) -> int | float:
    """Retourne la valeur absolue de x."""
    if est_positif(x):
```

```
        return x
    else:
        return -x
```

§4.1. Exercices de synthèse (Doc. 2,1 à Doc. 2,3)

⚠ Consigne

Écrire chaque fonction dans la partie principale de l'éditeur, en respectant sa spécification. Tester ensuite son appel dans la console.

Q27. Écrire le prédicat `est_majeur` dont la spécification est :

```python
def est_majeur(age: int) -> bool:
    """Retourne True si age >= 18, False sinon."""
```

Écrire ensuite, en réutilisant `est_majeur` et une f-string (Doc. 2,2), la fonction `message_age` dont la spécification est :

```python
def message_age(age: int) -> str:
    """Retourne un message indiquant si la personne est majeure,
    et sinon le nombre d'années avant sa majorité.

    >>> message_age(20)
    'Vous êtes majeur.'
    >>> message_age(15)
    'Vous êtes mineur, encore 3 an(s) avant la majorité.'
    """
```

Réponse :

```python
def est_majeur(age: int) -> bool:
    """Retourne True si age >= 18, False sinon."""
    return age >= 18

def message_age(age: int) -> str:
    """Retourne un message indiquant si la personne est majeure,
    et sinon le nombre d'années avant sa majorité.

    >>> message_age(20)
    'Vous êtes majeur.'
    >>> message_age(15)
    'Vous êtes mineur, encore 3 an(s) avant la majorité.'
    """
    if est_majeur(age):
```

```
        return f"Vous êtes majeur."
    else:
        return f"Vous êtes mineur, encore {18 - age} an(s) avant la majorité."
```

§5. Applications

Q28. Écrire la fonction (**avec sa spécification !**) `synthese`, qui attend une moyenne générale au BAC et retourne :

- `"Recalé"` si elle est strictement inférieure à 8 ;
- `"Second tour"` si elle est comprise entre 8 (inclus) et 10 (exclus) ;
- `"Admis"` si elle est comprise entre 10 (inclus) et 12 (exclus) ;
- `"Admis, Mention Assez Bien"` si elle est comprise entre 12 (inclus) et 14 (exclus) ;
- `"Admis, Mention Bien"` si elle est comprise entre 14 (inclus) et 16 (exclus) ;
- `"Admis, Mention Très Bien"` si elle est supérieure ou égale à 16.

Réponse :

```python
def synthese(moyenne: int | float) -> str:
    """Retourne la mention obtenue au BAC à partir de la moyenne générale.


    >>> synthese(7)
    'Recalé'
    >>> synthese(15)
    'Admis, Mention Bien'
    """
    if moyenne < 8:
        return "Recalé"
    elif moyenne < 10:
        return "Second tour"
    elif moyenne < 12:
        return "Admis"
    elif moyenne < 14:
        return "Admis, Mention Assez Bien"
    elif moyenne < 16:
        return "Admis, Mention Bien"
    else:
        return "Admis, Mention Très Bien"
```

Q29. Pour l'équation du second degré $ax^2 + bx + c = 0$ (avec a, b, c réels), l'arbre des cas possibles est le suivant :

- $a = 0$:
  - $b = 0$:
    - $c = 0$: tout réel est solution ;
    - $c \neq 0$: pas de solution ;
  - $b \neq 0$: une seule solution $x = -\frac{c}{b}$;
- $a \neq 0$, avec $\Delta = b^2 - 4ac$:

- $\Delta > 0$: deux solutions $x_1 = \frac{-b+\sqrt{\Delta}}{2a}$ et $x_2 = \frac{-b-\sqrt{\Delta}}{2a}$;
- $\Delta = 0$: une seule solution $x = -\frac{b}{2a}$;
- $\Delta < 0$: pas de solution réelle.

Écrire la fonction (**avec sa spécification !**) `racines`, à l'aide d'instructions `if`/`elif`/`else` imbriquées, qui retourne un message décrivant les solutions réelles.

Réponse :

```python
import math


def racines(a: int | float, b: int | float, c: int | float) -> str:
    """Retourne un message décrivant les solutions réelles de
    a * x**2 + b * x + c = 0.

    >>> racines(1, -3, 2)
    'deux solutions : x1 = 2.0, x2 = 1.0'
    >>> racines(1, 2, 1)
    'solution unique : x = -1.0'
    >>> racines(1, 0, 1)
    'pas de solution réelle'
    """
    if a == 0:
        if b == 0:
            if c == 0:
                return "tout réel est solution"
            else:
                return "pas de solution"
        else:
            x = -c / b
            return f"solution unique : x = {x}"
    else:
        delta = b**2 - 4 * a * c
        if delta > 0:
            x1 = (-b + math.sqrt(delta)) / (2 * a)
            x2 = (-b - math.sqrt(delta)) / (2 * a)
            return f"deux solutions : x1 = {x1}, x2 = {x2}"
        elif delta == 0:
            x = -b / (2 * a)
            return f"solution unique : x = {x}"
        else:
            return "pas de solution réelle"
```

Q30. Écrire la fonction (**avec sa spécification !**) `calcul_remise`, qui retourne le taux de remise (en %) applicable à un prix TTC :

— 0 % pour un montant TTC inférieur à 1000 € ;
— 1 % pour un montant TTC compris entre 1000 € (inclus) et 2000 € (exclus) ;
— 2 % pour un montant TTC compris entre 2000 € (inclus) et 5000 € (exclus) ;
— 5 % pour un montant TTC supérieur ou égal à 5000 €.

Écrire ensuite la fonction (**avec sa spécification !**) `calcul_prix`, qui retourne le prix final TTC (après remise) à partir d'un prix hors taxe, avec une TVA à 20 %. Cette fonction doit utiliser `calcul_remise`.

Réponse :

```python
def calcul_remise(prix_ttc: int | float) -> int:
    """Retourne le taux de remise (en %) applicable à un prix TTC.

    >>> calcul_remise(500)
    0
    >>> calcul_remise(1200)
    1
    """
    if prix_ttc >= 5000:
        return 5
    elif prix_ttc >= 2000:
        return 2
    elif prix_ttc >= 1000:
        return 1
    else:
        return 0

def calcul_prix(prix_ht: int | float) -> float:
    """Retourne le prix final TTC (après remise) à partir d'un prix HT,
    avec une TVA à 20 %.

    >>> calcul_prix(1000)
    1188.0
    """
    prix_ttc = prix_ht * 1.20
    remise = calcul_remise(prix_ttc)
    return prix_ttc * (1 - remise / 100)
```

Q31. Écrire le prédicat (**avec sa spécification !**) `est_bissextile`, qui détermine si une année est bissextile.

Indication : une année est bissextile si son millésime est multiple de 4 ; toutefois, un millésime multiple de 100 n'est bissextile que s'il est aussi multiple de 400 (1900 n'était pas bissextile, 2000 l'a été).

Réponse :

```python
def est_bissextile(n: int) -> bool:
```

```
    """Retourne True si l'année n est bissextile, False sinon.

    >>> est_bissextile(2024)
    True
    >>> est_bissextile(1900)
    False
    >>> est_bissextile(2000)
    True
    """
    return n % 4 == 0 and (n % 100 != 0 or n % 400 == 0)
```

Q32. On rappelle l'**inégalité triangulaire** : trois longueurs `a`, `b`, `c` peuvent former un triangle si et seulement si chaque côté est strictement inférieur à la somme des deux autres. Écrire le prédicat (**avec sa spécification !**) `est_triangle`, qui doit combiner **trois** conditions avec `and` :

```python
def est_triangle(a: int | float, b: int | float, c: int | float) -> bool:
    """Retourne True si a, b, c peuvent être les longueurs des côtés
    d'un triangle, False sinon.

    >>> est_triangle(3, 4, 5)
    True
    >>> est_triangle(1, 2, 5)
    False
    """
```

Réponse :

```python
def est_triangle(a: int | float, b: int | float, c: int | float) -> bool:
    """Retourne True si a, b, c peuvent être les longueurs des côtés
    d'un triangle, False sinon.

    >>> est_triangle(3, 4, 5)
    True
    >>> est_triangle(1, 2, 5)
    False
    """
    return a < b + c and b < a + c and c < a + b
```

Q33. Écrire la fonction (**avec sa spécification !**) `type_triangle`, qui doit utiliser `est_triangle` et retourner :

- `"pas un triangle"` si `a`, `b`, `c` ne peuvent pas former un triangle ;
- `"équilatéral"` si les trois côtés sont égaux ;
- `"isocèle"` si exactement deux côtés sont égaux ;
- `"scalène"` sinon (aucun côté égal à un autre).

```python
>>> type_triangle(1, 2, 5)
'pas un triangle'
>>> type_triangle(3, 3, 3)
'équilatéral'
>>> type_triangle(3, 3, 5)
'isocèle'
>>> type_triangle(3, 4, 5)
'scalène'
```

Réponse :

```python
def type_triangle(a: int | float, b: int | float, c: int | float) -> str:
    """Retourne le type d'un triangle de côtés a, b, c.

    >>> type_triangle(1, 2, 5)
    'pas un triangle'
    >>> type_triangle(3, 3, 3)
    'équilatéral'
    >>> type_triangle(3, 3, 5)
    'isocèle'
    >>> type_triangle(3, 4, 5)
    'scalène'
    """
    if not est_triangle(a, b, c):
        return "pas un triangle"
    elif a == b == c:
        return "équilatéral"
    elif a == b or b == c or a == c:
        return "isocèle"
    else:
        return "scalène"
```

Bilan : où en suis-je ?

Pour chaque ligne, cochez la colonne qui correspond. Toute case **Pas encore** indique une partie à reprendre.

Je sais…	Oui	Pas encore	À revoir
distinguer une **affectation** (=) d'une **comparaison** (==)			Partie 1
utiliser les opérateurs de comparaison (<, >=, !=, …)			Partie 1
combiner des conditions avec `and`, `or`, `not`			Partie 2
expliquer pourquoi Python n'évalue pas toujours les deux opérandes d'un `and` ou d'un `or`			Partie 2

écrire une **alternative** (`if`, `if`/`else`, `if`/`elif`/`else`)			Partie 3
écrire un **prédicat** qui réutilise un autre prédicat déjà écrit			Partie 4