

TP 3.1 – Premiers algorithmes récursifs

Objectifs –

- ◇ Reconnaître le cas de base d'une fonction récursive.
- ◇ Formuler un appel récursif portant sur un argument plus simple.
- ◇ Dérouler les appels d'une fonction récursive.
- ◇ Comparer des formulations récursive et itérative.

Parcours principal

Exercice 1 – Factorielle récursive, le retour

Pour tout entier naturel n , la factorielle de n est définie par :

$$0! = 1 \quad \text{et} \quad n! = n \times (n-1)! \quad \text{si } n \geq 1$$

Question 1. Réécrire, en respectant les bonnes pratiques, la fonction récursive `factorielle(n: int) -> int` vue en cours. On précisera sa précondition.

Question 2. Dérouler l'appel `factorielle(4)` en indiquant les appels récursifs successifs, puis les valeurs renvoyées.

Question 3. Tester la fonction sur plusieurs valeurs bien choisies.

Exercice 2 – Dans les étoiles

On considère les deux fonctions récursives suivantes :

```
def etoiles_decroissantes(n: int) -> None:
    assert n >= 0
    if n > 0:
        print("*" * n)
        etoiles_decroissantes(n - 1)

def etoiles_croissantes(n: int) -> None:
    assert n >= 0
    if n > 0:
        etoiles_croissantes(n - 1)
```

```
print("*" * n)
```

Question 1. Sans exécuter le programme, prévoir ce que produit l'appel `etoiles_decroissantes(5)`, puis vérifier la réponse.

Question 2. Faire de même avec l'appel `etoiles_croissantes(5)`.

Question 3. Expliquer la différence entre les deux affichages. Quel est le type de la valeur renvoyée par ces fonctions ?

Exercice 3 – Une suite récurrente

On considère la suite $(u_n)_{n \in \mathbb{N}}$ définie par :

$$u_0 = 2 \quad \text{et} \quad u_{n+1} = \frac{1}{2} \left(u_n + \frac{3}{u_n} \right)$$

Question 1. Calculer à la main u_1 , u_2 et u_3 .

Question 2. Écrire une fonction récursive `terme_u_rec(n: int) -> float` qui renvoie u_n .

Question 3. Écrire une fonction itérative `terme_u_iter(n: int) -> float` ayant la même spécification.

Question 4. Tester les deux fonctions et vérifier qu'elles donnent les mêmes résultats. Vers quel nombre les premiers termes semblent-ils converger ?

Exercice 4 – Exponentiation naïve

Pour tout réel x et tout entier naturel n , on utilise la définition récursive suivante :

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ x \times x^{n-1} & \text{si } n \geq 1 \end{cases}$$

Question 1. Lister les appels récursifs successifs engendrés par `exp_naive(2, 5)`.

Question 2. Écrire une fonction récursive `exp_naive(x: float, n: int) -> float` fondée sur cette définition.

Question 3. Tester la fonction. Combien de multiplications et d'appels récursifs sont nécessaires pour calculer x^n lorsque $n > 0$?

Si le temps le permet

Exercice 5 – Exponentiation rapide

Soit n un entier naturel. En posant $q = n // 2$, on obtient :

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ (x^2)^q & \text{si } n = 2q > 0 \\ x(x^2)^q & \text{si } n = 2q + 1 \end{cases}$$

Cette formulation permet de diviser l'exposant par 2 à chaque appel récursif.

Question 1. Lister les appels récursifs engendrés par `exp_rapide(2, 10)` et comparer leur nombre avec celui de la méthode naïve.

Question 2. Écrire une fonction récursive `exp_rapide(x: float, n: int) -> float` fondée sur cette définition.

Question 3. Valider la fonction par plusieurs tests.

Travail à la maison

Exercice 6 – Deux autres suites récurrentes

On considère les suites (v_n) et (w_n) définies par :

$$v_0 = 1 \quad \text{et} \quad v_{n+1} = 2v_n + n$$

et par :

$$w_0 = 1, \quad w_1 = 0, \quad w_{n+2} = w_{n+1}^2 + 2w_n$$

Question 1. Écrire une fonction récursive `terme_v(n: int) -> int` qui renvoie v_n .

Question 2. Écrire une fonction récursive naïve `terme_w(n: int) -> int` qui renvoie w_n .

Question 3. Calculer les six premiers termes de chacune des suites et les utiliser pour tester les fonctions.

Question 4. Pourquoi la fonction `terme_w` risque-t-elle d'effectuer plusieurs fois les mêmes calculs ?

Approfondissement facultatif

Exercice 7 – Limites pratiques de la récursivité

Python fixe une profondeur maximale pour les appels récursifs. On peut afficher la limite courante avec :

```
import sys
print("Limite de récursion:", sys.getrecursionlimit())
```

Question 1. Essayer d'observer cette limite en appelant `factorielle` avec des valeurs de plus en plus grandes. Quelle exception obtient-on ?

Question 2. Lorsqu'on tente d'afficher une très grande factorielle, Python peut également produire le message suivant :

```
ValueError: Exceeds the limit (4300 digits) for integer
string conversion
```

Cette erreur a-t-elle la même origine que la précédente ?

Corrigés des exercices

Corrigé – Exercice 1 – Factorielle récursive, le retour

Question 1.

```
def factorielle(n: int) -> int:
    """Renvoie la factorielle de n.

    Précondition: n >= 0.
    """
    assert n >= 0
    if n == 0:
        return 1
    return n * factorielle(n - 1)
```

Question 2. Les appels récursifs sont :

```
factorielle(4)
factorielle(3)
factorielle(2)
factorielle(1)
factorielle(0)
```

Le dernier appel renvoie 1, puis les appels en attente renvoient successivement 1, 2, 6 et 24. Ainsi :

$$4! = 4 \times 3 \times 2 \times 1 = 24$$

Question 3.

```
assert factorielle(0) == 1
assert factorielle(1) == 1
assert factorielle(4) == 24
assert factorielle(10) == 3628800
```

Corrigé – Exercice 2 – Dans les étoiles

Question 1. Le premier appel produit :

```
*****
```

```
****
***
**
*
```

Question 2. Le second appel produit :

```
*
**
***
****
*****
```

Question 3. Dans la première fonction, l'affichage est effectué avant l'appel récursif. Dans la seconde, il est effectué lorsque les appels récursifs se terminent, dans l'ordre inverse de leur lancement. Les deux fonctions ne renvoient explicitement aucune valeur : elles renvoient donc **None**, de type **NoneType**.

Corrigé – Exercice 3 – Une suite récurrente

Question 1. On obtient :

$$u_1 = \frac{7}{4}, \quad u_2 = \frac{97}{56}, \quad u_3 = \frac{18817}{10864}$$

Les valeurs décimales correspondantes sont environ 1,75, 1,73214 et 1,73205.

Question 2.

```
def terme_u_rec(n: int) -> float:
    """Renvoie le terme u_n.

    Précondition: n >= 0.
    """
    assert n >= 0
    if n == 0:
        return 2.0
    precedent = terme_u_rec(n - 1)
    return (precedent + 3 / precedent) / 2
```

La variable **precedent** évite de calculer deux fois le même terme.

Question 3.

```
def terme_u_iter(n: int) -> float:
    """Renvoie le terme u_n.

    Précondition: n >= 0.
    """
    assert n >= 0
    u = 2.0
    for k in range(n):
        u = (u + 3 / u) / 2
    return u
```

Question 4. Les nombres flottants sont comparés avec une tolérance :

```
for n in range(10):
    assert abs(terme_u_rec(n) - terme_u_iter(n)) < 1e-12

print(terme_u_rec(5))
```

Les premiers termes semblent converger vers $\sqrt{3}$.

Corrigé – Exercice 4 – Exponentiation naïve

Question 1. Les appels successifs sont :

```
exp_naive(2, 5)
exp_naive(2, 4)
exp_naive(2, 3)
exp_naive(2, 2)
exp_naive(2, 1)
exp_naive(2, 0)
```

Ils renvoient ensuite successivement 1, 2, 4, 8, 16 et 32.

Question 2.

```
def exp_naive(x: float, n: int) -> float:
    """Renvoie x à la puissance n.

    Précondition: n >= 0.
    """
    assert n >= 0
```

```
if n == 0:
    return 1.0
return x * exp_naive(x, n - 1)
```

Question 3.

```
assert exp_naive(2, 0) == 1
assert exp_naive(2, 5) == 32
assert exp_naive(-3, 3) == -27
assert exp_naive(0.5, 4) == 0.0625
```

Le calcul de x^n nécessite n appels récursifs, auxquels s'ajoute l'appel initial, et n multiplications.

Corrigé – Exercice 5 – Exponentiation rapide

Question 1. Les couples d'arguments successifs sont :

```
(2, 10), (4, 5), (16, 2), (256, 1), (65536, 0)
```

Il n'y a que quatre appels récursifs, contre dix avec la version naïve.

Question 2.

```
def exp_rapide(x: float, n: int) -> float:
    """Renvoie x à la puissance n par exponentiation rapide.

    Précondition: n >= 0.
    """
    assert n >= 0
    if n == 0:
        return 1.0
    puissance = exp_rapide(x * x, n // 2)
    if n % 2 == 0:
        return puissance
    return x * puissance
```

Question 3.

```
for x in [-2, -0.5, 0, 1.5, 3]:
    for n in range(12):
```

```
assert abs(exp_rapide(x, n) - x**n) < 1e-12
```

Lorsque les calculs portent sur des nombres flottants, il est préférable de faire une comparaison avec une tolérance.

Corrigé – Exercice 6 – Deux autres suites récurrentes

Question 1. Pour calculer v_n , l'appel récursif fournit v_{n-1} :

```
def terme_v(n: int) -> int:
    """Renvoie le terme v_n.

    Précondition: n >= 0.
    """
    assert n >= 0
    if n == 0:
        return 1
    return 2 * terme_v(n - 1) + n - 1
```

Question 2.

```
def terme_w(n: int) -> int:
    """Renvoie le terme w_n par une méthode récursive naïve.

    Précondition: n >= 0.
    """
    assert n >= 0
    if n == 0:
        return 1
    if n == 1:
        return 0
    return terme_w(n - 1)**2 + 2 * terme_w(n - 2)
```

Question 3. On obtient :

$$(v_0, v_1, v_2, v_3, v_4, v_5) = (1, 2, 5, 12, 27, 58)$$

et :

$$(w_0, w_1, w_2, w_3, w_4, w_5) = (1, 0, 2, 4, 20, 408)$$

On peut donc écrire :

```
assert [terme_v(n) for n in range(6)] \
== [1, 2, 5, 12, 27, 58]
assert [terme_w(n) for n in range(6)] \
== [1, 0, 2, 4, 20, 408]
```

Question 4. Pour calculer $\text{terme_w}(n)$, la fonction lance deux appels récursifs. Chacun d'eux engendre à son tour deux appels : un même terme est donc recalculé de nombreuses fois. Cette formulation naïve devient rapidement inefficace.

Corrigé – Exercice 7 – Limites pratiques de la récursivité

Question 1. Lorsque la profondeur maximale est dépassée, Python déclenche une exception de type `RecursionError`. Cette limite protège l'interpréteur contre un débordement de la pile d'appels.

Question 2. La seconde erreur est différente. Python a réussi à calculer l'entier, mais refuse de le convertir en une chaîne comportant un nombre de chiffres supérieur à la limite de sécurité. Elle peut donc apparaître au moment de l'affichage d'une grande factorielle, sans que la profondeur maximale de récursion ait été atteinte.