Suivre l'évolution des variables

Doc. 2,6

Objectif

Faire tourner des programmes à la main, en notant dans un tableau l'évolution du contenu des variables, pour prévoir ce que font un appel de fonction, une alternative et une boucle.

Chaque exercice se fait d'abord sur papier, sans ordinateur. Le programme est ensuite recopié dans Thonny et exécuté en mode débogage (touche F7) pour vérifier le tableau.

Méthode : remplir un tableau d'état

- Une ligne du tableau correspond à une instruction exécutée, dans l'ordre où Python l'exécute (ou à une itération, quand le tableau le précise).
- Sur chaque ligne, on écrit la valeur de chaque variable juste après l'exécution de l'instruction. Une case reste vide tant que la variable n'existe pas.
- Pour une condition (`if`, `elif`, `while`), on la recopie en remplaçant les variables par leurs valeurs, puis on écrit sa valeur : `True` ou `False`.
- Pour calculer une expression, on utilise les valeurs que les variables ont à ce moment-là, et non leurs valeurs du début.

§1. Exercice 1 — Suivre un appel de fonction

On considère le programme suivant.

```python
def transforme(x: int) -> int:
    """Fonction à étudier."""
    y = 2 * x
    x = y + 1
    return x * y

a = 3
b = transforme(a)
c = transforme(b - 40)
```

Q1. Sans utiliser Python, compléter le tableau d'état ci-dessous. Les variables `x` et `y` n'existent que pendant l'exécution de la fonction : leurs cases restent vides en dehors d'un appel.

Ligne	a	b	c	x	y
7					
8 : appel de `transforme`					
3					
4					

5 : valeur retournée					
8 : affectation de `b`					
9 : appel de `transforme`					
3					
4					
5 : valeur retournée					
9 : affectation de `c`					

Réponse :

Ligne	**a**	**b**	**c**	**x**	**y**
7	3				
8 : appel de `transforme(3)`	3			3	
3	3			3	6
4	3			7	6
5 : retourne `42`	3			7	6
8 : affectation de `b`	3	42			
9 : appel de `transforme(2)`	3	42		2	
3	3	42		2	4
4	3	42		5	4
5 : retourne `20`	3	42		5	4
9 : affectation de `c`	3	42	20		

À la ligne 9, l'argument `b - 40` est évalué avant l'appel : il vaut `2`. C'est cette valeur que reçoit le paramètre `x`. À la fin, `a = 3`, `b = 42` et `c = 20`.

Q2. À la ligne 4, la valeur de `x` change. Pourquoi `a` vaut-elle toujours `3` à la fin du programme ?

Réponse : Lors de l'appel `transforme(a)`, le paramètre `x` reçoit la valeur de l'argument, c'est-à-dire `3`. Ensuite, `x` et `a` sont deux variables différentes : affecter une nouvelle valeur à `x` ne modifie pas `a`.

Q3. Sans utiliser Python, déterminer la valeur de `transforme(transforme(1))` en détaillant les étapes. Vérifier ensuite dans la console.

Réponse : Python évalue d'abord l'argument `transforme(1)` : `y` prend la valeur `2`, puis `x` la valeur `3` ; l'appel retourne `3 * 2`, soit `6`.

L'appel se ramène alors à `transforme(6)` : `y` vaut `12`, `x` vaut `13` ; l'appel retourne `13 * 12`, soit `156`.

§2. Exercice 2 — Une fonction qui en appelle une autre

Dans le programme suivant, trois variables différentes portent le même nom `n` : celle du programme principal, le paramètre de `f` et le paramètre de `double`.

```python
def double(n: int) -> int:
    """Retourne le double de n."""
    n = 2 * n
    return n

def f(n: int) -> int:
    """Fonction à étudier."""
    m = double(n) + 1
    n = double(m)
    return n - m

n = 3
r = f(n)
s = f(r) + n
```

Q4. Sans utiliser Python, compléter le tableau d'état ci-dessous pour les lignes 12 et 13 du programme. La colonne « En cours » indique où se trouve l'exécution : programme principal, `f` ou `double`.

Ligne	**En cours**	**n** (programme)	**r**	**n** (dans `f`)	**m**	**n** (dans `double`)
12	programme	3				

Réponse :

Ligne	**En cours**	**n** (programme)	**r**	**n** (dans `f`)	**m**	**n** (dans `double`)
12	programme	3				
13 : appel `f(3)`	`f`	3		3		
8 : appel `double(3)`	`double`	3		3		3
3	`double`	3		3		6

4 : retourne 6	double	3		3		6
8 : affectation de m	f	3		3	7	
9 : appel double(7)	double	3		3	7	7
3	double	3		3	7	14
4 : retourne 14	double	3		3	7	14
9 : affectation de n	f	3		14	7	
10 : retourne 7	f	3		14	7	
13 : affectation de r	programme	3	7			

Le n de f passe de 3 à 14, celui de double de 3 à 6 : aucun des deux ne modifie le n du programme principal, qui vaut toujours 3.

Q5. Sans construire de nouveau tableau, déterminer la valeur de s après la ligne 14. Quelle valeur de n est utilisée dans l'expression f(r) + n ?

Réponse : r vaut 7. Dans l'appel f(7) : double(7) retourne 14, donc m vaut 15 ; double(15) retourne 30, donc le n de f vaut 30 ; l'appel retourne 30 − 15, soit 15.

Dans f(r) + n, on est dans le programme principal : c'est son n qui est utilisé, et il vaut 3. Donc s = 15 + 3 = 18.

Q6. Un élève remplace les lignes 3 et 4 par la seule instruction print(2 * n). Prévoir ce qui se passe lors de l'exécution de la ligne 13, puis vérifier.

Réponse : L'appel double(3) affiche 6 mais ne retourne aucune valeur : il retourne None. La ligne 8 doit alors calculer None + 1, ce qui provoque une erreur TypeError. Afficher une valeur n'est pas la retourner : seule une valeur retournée peut être réutilisée dans un calcul.

§3. Exercice 3 — Quelle branche est exécutée ?

Un cinéma calcule le prix d'une place avec la fonction suivante.

Python

```
1  AGE_ENFANT = 12
2  AGE_SENIOR = 65
3  PRIX_ENFANT = 5
4  PRIX_REDUIT = 7
5  PRIX_NORMAL = 10
6
7  def tarif(age: int, etudiant: bool) -> int:
8      """Retourne le prix de la place, en euros."""
9      if age < AGE_ENFANT:
10         prix = PRIX_ENFANT
11     elif age >= AGE_SENIOR or etudiant:
12         prix = PRIX_REDUIT
13     else:
14         prix = PRIX_NORMAL
```

```
15      return prix
```

Q7. Sans utiliser Python, compléter le tableau ci-dessous. Dans la colonne du milieu, écrire chaque condition évaluée, dans l'ordre, avec les valeurs des variables et le résultat de l'évaluation. Une condition qui n'est pas évaluée ne doit pas apparaître.

Appel	Conditions évaluées	Valeur retournée
`tarif(8, True)`		
`tarif(70, False)`		
`tarif(17, True)`		
`tarif(30, False)`		
`tarif(12, False)`		

Réponse :

Appel	Conditions évaluées	Valeur retournée
`tarif(8, True)`	`8 < 12` : `True`	`5`
`tarif(70, False)`	`70 < 12` : `False` `70 >= 65 or ...` : `True`	`7`
`tarif(17, True)`	`17 < 12` : `False` `17 >= 65 or True` : `True`	`7`
`tarif(30, False)`	`30 < 12` : `False` `30 >= 65 or False` : `False`	`10`
`tarif(12, False)`	`12 < 12` : `False` `12 >= 65 or False` : `False`	`10`

Pour `tarif(70, False)`, `70 >= 65` vaut déjà `True` : grâce à l'évaluation en court-circuit, Python n'a pas besoin d'évaluer `etudiant`. Pour `tarif(12, False)`, `12 < 12` est fausse : l'inégalité est stricte.

Q8. Une personne de 8 ans est aussi étudiante dans une école de musique : `tarif(8, True)` retourne `5` et non `7`. Expliquer pourquoi à partir de la structure `if` / `elif` / `else`.

Réponse : Dans une structure `if` / `elif` / `else`, un seul bloc est exécuté : celui de la première condition vraie. Ici, `8 < 12` est vraie, donc `prix` prend la valeur `5` et la condition du `elif` n'est même pas évaluée.

Q9. Un élève remplace `elif` par `if` à la ligne 11. Le programme reste correct pour Python. Sans utiliser Python, déterminer ce que retournent alors `tarif(8, True)` et `tarif(8, False)`. Vérifier ensuite.

Réponse : Le programme contient maintenant deux alternatives indépendantes, exécutées l'une après l'autre : un `if` seul (lignes 9 et 10), puis un `if` / `else` (lignes 11 à 14).

- `tarif(8, True)` : `8 < 12` est vraie, `prix` vaut `5`. Puis `8 >= 65 or True` est vraie, `prix` vaut `7`. L'appel retourne `7`.
- `tarif(8, False)` : `prix` vaut d'abord `5`. Puis `8 >= 65 or False` est fausse : c'est le bloc `else` qui s'exécute et `prix` vaut `10`. L'appel retourne `10`, le prix normal, pour un enfant.

§4. Exercice 4 — Des if successifs ou un elif ?

On compare deux versions d'une fonction de calcul de score. En dehors de leur nom, elles ne diffèrent que par les mots-clés des lignes 5 et 7.

```python
def ajuste_a(score: int) -> int:
    """Calcule un nouveau score."""
    if score < 10:
        score = score + 3
    if score < 15:
        score = score + 2
    if score > 14:
        score = score + 1
    return score
```

```python
def ajuste_b(score: int) -> int:
    """Calcule un nouveau score."""
    if score < 10:
        score = score + 3
    elif score < 15:
        score = score + 2
    elif score > 14:
        score = score + 1
    return score
```

Q10. Sans utiliser Python, suivre l'appel `ajuste_a(9)` en complétant le tableau ci-dessous. Chaque ligne du tableau correspond à une ligne exécutée de la fonction. Toutes les lignes du tableau ne seront pas forcément utilisées.

Ligne	Condition testée	score
1		9

Réponse :

Ligne	Condition testée	score
1		9
3	`9 < 10` : `True`	9
4		12
5	`12 < 15` : `True`	12
6		14
7	`14 > 14` : `False`	14
9		14

L'appel retourne `14`. La condition de la ligne 5 utilise la valeur de `score` déjà modifiée à la ligne 4 (`12`), et non la valeur de départ (`9`).

Q11. Faire de même pour `ajuste_b(9)`, puis compléter le tableau des valeurs retournées ci-dessous.

score de départ	ajuste_a	ajuste_b

9	14	
13		
16		

Réponse : Pour `ajuste_b(9)`, les lignes exécutées sont 1, 3 (`9 < 10` : `True`), 4 (`score` vaut `12`) puis 9. Les deux `elif` ne sont pas évalués : l'appel retourne `12`.

score de départ	ajuste_a	ajuste_b
9	14	12
13	16	15
16	17	17

Pour `13` : dans la version A, `13 < 15` est vraie et `score` devient `15` ; puis `15 > 14` est vraie et `score` devient `16`. Dans la version B, seul le premier bloc dont la condition est vraie est exécuté : `score` devient `15`.

Q12. **Plus difficile.** Pour quelles valeurs entières de `score`, comprises entre 0 et 20, les deux versions retournent-elles des valeurs différentes ? Justifier, puis vérifier avec une boucle dans la console.

Réponse : Dans la version B, un seul bloc est exécuté. Dans la version A, un bloc peut être exécuté puis modifier `score` de façon qu'une condition suivante devienne vraie.

- De 0 à 9 : la version A ajoute 3 puis 2 (le score obtenu est inférieur à 15), alors que la version B ajoute seulement 3. Les résultats diffèrent.
- De 10 à 12 : les deux versions ajoutent 2. Dans la version A, le score obtenu (12 à 14) ne dépasse pas 14 : même résultat.
- 13 et 14 : les deux versions ajoutent 2, mais dans la version A le score obtenu (15 ou 16) dépasse 14 et on ajoute encore 1. Les résultats diffèrent.
- De 15 à 20 : seule la condition `score > 14` est vraie dans les deux versions : même résultat.

Les résultats diffèrent donc pour les scores de 0 à 9, 13 et 14. Vérification :

```python
>>> for score in range(21):
...     if ajuste_a(score) != ajuste_b(score):
...         print(score, end=" ")
0 1 2 3 4 5 6 7 8 9 13 14
```

§5. Exercice 5 — Faire tourner une boucle à la main

On considère les deux fonctions suivantes.

```python
def mystere(n: int) -> int:
    """Fonction à étudier."""
    s = 0
    for i in range(1, n + 1, 2):
        s = s + i
    return s
```

```python
def mystere_bis(n: int) -> int:
    """Fonction à étudier."""
    s = 0
    i = 1
    while i <= n:
        s = s + i
        i = i + 2
    return s
```

Q13. Quels entiers sont produits par `range(1, 8, 2)` ? Combien d'itérations la boucle de `mystere(7)` effectue-t-elle ?

Réponse : `range(1, 8, 2)` produit `1`, `3`, `5` et `7` : on part de 1, on avance de 2, et 8 est exclu. La boucle effectue donc 4 itérations.

Q14. Sans utiliser Python, compléter le tableau d'état de l'appel `mystere(7)`. Chaque ligne donne les valeurs à la fin d'une itération.

Étape	i	s
avant la boucle		0
itération 1		
itération 2		
itération 3		
itération 4		

Réponse :

Étape	i	s
avant la boucle		0
itération 1	1	1
itération 2	3	4
itération 3	5	9
itération 4	7	16

L'appel retourne `16`. Avant la boucle, `i` n'existe pas encore : c'est la boucle `for` qui lui donne ses valeurs.

Q15. Sans utiliser Python, compléter le tableau d'état de l'appel `mystere_bis(7)`. La condition de la boucle est évaluée au début de chaque itération : la dernière ligne correspond au test qui arrête la boucle.

Étape	Condition testée	s	i
avant la boucle		0	1
itération 1	`1 <= 7` : `True`		
itération 2			
itération 3			

itération 4			
fin			

Réponse :

Étape	Condition testée	s	i
avant la boucle		0	1
itération 1	`1 <= 7` : `True`	1	3
itération 2	`3 <= 7` : `True`	4	5
itération 3	`5 <= 7` : `True`	9	7
itération 4	`7 <= 7` : `True`	16	9
fin	`9 <= 7` : `False`	16	9

L'appel retourne `16`, comme `mystere(7)`.

Q16. À la fin de chaque appel, quelle est la dernière valeur prise par `i` : dans `mystere(7)` ? dans `mystere_bis(7)` ? Expliquer la différence.

Réponse : Dans `mystere(7)`, `i` vaut `7` : c'est la dernière valeur produite par `range`. Dans `mystere_bis(7)`, `i` vaut `9` : la boucle `while` ne s'arrête que lorsque la condition devient fausse, donc après que `i` a dépassé `7`.

Q17. Sans utiliser Python, prévoir les valeurs de `mystere(8)` et de `mystere_bis(0)`. Combien d'itérations sont effectuées dans le second appel ?

Réponse : `range(1, 9, 2)` produit les mêmes entiers que `range(1, 8, 2)` : `mystere(8)` retourne aussi `16`.

Pour `mystere_bis(0)`, la condition `1 <= 0` est fausse dès le premier test : aucune itération n'est effectuée et l'appel retourne `0`.

Q18. Quel calcul réalisent ces deux fonctions ?

Réponse : Elles retournent la somme des entiers impairs compris entre 1 et `n`. On remarque que les valeurs successives de `s` (1, 4, 9, 16) sont des carrés.

§6. Exercice 6 — Fonctions, alternative et boucles réunies

§6.1. Une boucle non bornée qui appelle une fonction

On part d'un entier strictement positif `n`. S'il est pair, on le divise par 2 ; sinon, on le multiplie par 3 et on ajoute 1. On recommence jusqu'à obtenir 1. La fonction `syracuse` compte le nombre d'étapes et retient la plus grande valeur atteinte.

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

```

```python
def syracuse(n: int) -> str:
    """Retourne le nombre d'étapes pour atteindre 1
    et la plus grande valeur atteinte.
    """
    etapes = 0
    maxi = n
    while n != 1:
        if est_pair(n):
            n = n // 2
        else:
            n = 3 * n + 1
        if n > maxi:
            maxi = n
        etapes += 1
    return f"{etapes} étapes, maximum {maxi}"
```

Q19. Sans utiliser Python, compléter le tableau d'état de l'appel `syracuse(6)`. Chaque ligne donne les valeurs à la fin d'une itération ; la colonne `est_pair(n)` contient la valeur retournée par cet appel, au début de l'itération. Toutes les lignes ne seront pas forcément utilisées. Quelle chaîne de caractères est retournée ?

Étape	`est_pair(n)`	n	maxi	etapes
avant la boucle		6		
itération 1				
itération 2				
itération 3				
itération 4				
itération 5				
itération 6				
itération 7				
itération 8				
itération 9				
itération 10				

Réponse :

Étape	`est_pair(n)`	n	maxi	etapes
avant la boucle		6	6	0
itération 1	`True`	3	6	1
itération 2	`False`	10	10	2
itération 3	`True`	5	10	3
itération 4	`False`	16	16	4

itération 5	`True`	8	16	5
itération 6	`True`	4	16	6
itération 7	`True`	2	16	7
itération 8	`True`	1	16	8

Après l'itération 8, la condition `1 != 1` est fausse : la boucle s'arrête. L'appel retourne `'8 étapes, maximum 16'`. À l'itération 1, `3 > 6` est fausse : `maxi` garde la valeur `6`.

Q20. Sans utiliser Python, prévoir la valeur retournée par `syracuse(1)`. Combien de fois le bloc de la boucle est-il exécuté ?

Réponse : `maxi` prend la valeur `1` et `etapes` la valeur `0`. La condition `1 != 1` est fausse dès le premier test : le bloc n'est exécuté aucune fois. L'appel retourne `'0 étapes, maximum 1'`.

Q21. Un élève appelle `syracuse(0)`, alors que la fonction attend un entier strictement positif. Faire tourner à la main les deux premières itérations. Que se passe-t-il ? Quelle modification de la ligne 11 éviterait ce problème ?

Réponse : `0 != 1` est vraie ; `est_pair(0)` retourne `True`, donc `n` prend la valeur `0 // 2`, soit `0`. À l'itération suivante, la situation est exactement la même : `n` vaut toujours `0` et ne se rapproche jamais de 1. C'est une boucle infinie.

Avec `while n > 1:`, la condition `0 > 1` est fausse dès le premier test et la boucle n'est pas exécutée. Pour un entier strictement positif, les deux conditions donnent le même résultat.

§6.2. Deux boucles imbriquées

```python
def motif(n: int) -> str:
    """Fonction à étudier."""
    texte = ""
    for i in range(1, n + 1):
        for j in range(i):
            texte += "*"
        texte += "-"
    return texte
```

Q22. Sans utiliser Python, compléter le tableau d'état de l'appel `motif(3)`. Une ligne correspond à une instruction `texte += ...` exécutée ; la colonne `j` reste vide pour l'instruction `texte += "-"`. Quelle chaîne est retournée ?

i	j	texte

Réponse :

i	j	texte
1	0	"*"
1		"*-"
2	0	"*-*"
2	1	"*-**"
2		"*-**-"
3	0	"*-**-*"
3	1	"*-**-**"
3	2	"*-**-***"
3		"*-**-***-"

L'appel retourne `'*-**-***-'`. Pour chaque valeur de `i`, la boucle interne `for j in range(i)` effectue `i` itérations : elle ajoute `i` étoiles, puis l'instruction `texte += "-"`, qui n'est pas dans la boucle interne, ajoute un tiret.

Q23. Sans utiliser Python, déterminer le nombre de caractères de la chaîne retournée par `motif(4)`. Vérifier avec `len(motif(4))`.

Réponse : `motif(4)` contient $1 + 2 + 3 + 4 = 10$ étoiles et 4 tirets, soit 14 caractères.

§7. Bilan

À l'issue de cette activité, je sais…

- remplir un tableau d'état en utilisant, pour chaque instruction, les valeurs des variables juste avant son exécution ;
- suivre un appel de fonction : le paramètre reçoit la valeur de l'argument, les variables de la fonction disparaissent après le `return` ;
- expliquer pourquoi modifier un paramètre ne modifie pas la variable passée en argument ;
- distinguer une fonction qui retourne une valeur d'une fonction qui l'affiche ;
- repérer le seul bloc exécuté dans une structure `if` / `elif` / `else`, et la différence avec des `if` successifs ;
- compter les itérations d'une boucle `for` avec `range` et d'une boucle `while`, y compris lorsqu'il n'y en a aucune ;
- repérer une boucle infinie en faisant tourner une boucle à la main ;
- suivre deux boucles imbriquées.