Version pdf

Chapitre 6 : fonctions

"Objectifs"

À la fin du chapitre, on doit savoir :

- écrire une fonction avec ou sans paramètres, avec ou sans `return`
- écrire la spécification d'une fonction dans une docstring
- comprendre les annotations de type
- distinguer `return` et `print`
- importer correctement un module Python dont on veut utiliser une fonction
- comprendre la distinction entre portée locale et globale d'une fonction
- écrire un jeu de tests unitaires pour une fonction bien spécifiée

Fonction avec un seul paramètre

"Exemple 1"

En mathématiques, on peut définir la fonction carré par $f : x \mapsto x^2$.

En Python, on définit une fonction avec le mot-clé `def` .

```
def f(x):
    return x ** 2
```

Dans les deux cas, `x` est l'entrée de la fonction. L'appel de fonction s'écrit de la même manière en mathématiques et en Python : `f(3)` . Cet appel renvoie la valeur `9` .

On distingue donc deux étapes :

- la **définition** de la fonction ;
- l'**appel** de la fonction avec une valeur donnée.

"Définition 1"

Une **fonction** est un bloc de code auquel on donne un nom et que l'on peut réutiliser.

Une fonction peut recevoir une ou plusieurs valeurs en entrée, appelées **paramètres**.

Une fonction peut renvoyer une valeur avec l'instruction `return` .

En Python, une fonction est définie avec le mot clef `def` suivi de son nom et de la liste éventuellement vide de ses paramètres. Ensuite vient Le bloc de code qui est **indenté**.

```
def fonction(parametre):
    # bloc de code indenté
    traitement
    return valeur
```

Un paramètre est utilisé dans le corps de la fonction comme une **variable locale**. Il n'existe pas en dehors de la fonction.

L'**appel** de la fonction est effectué en donnant aux paramètres des valeurs. Il décleinche l'exécution du bloc de code qui se termine soit par une instruction finale soit dès le premier `return` qui renvoie une valeur.

Une fonction doit être accompagnée d'une **spécification**, placée dans une chaîne de caractères appelée **docstring**.

La spécification précise :

- le rôle de la fonction ;
- le type attendu pour les paramètres ;
- le type de la valeur renvoyée ;
- les éventuelles conditions d'utilisation.

Exemple :

```
def carre(x: float) -> float:
    """
    Prend en paramètre un nombre x de type float.
    Renvoie le carré de x.
    """
    return x ** 2
```

Les annotations `x: float` et `-> float` indiquent que le paramètre `x` est attendu de type `float` et que la valeur renvoyée est aussi de type `float`.

Ces annotations aident à lire le programme mais ne sont pas vérifiées automatiquement par Python pendant l'exécution.

Si une docstring a été écrite, on peut l'afficher dans la console avec la fonction `help`, par exemple `help(carre)`.

"Remarque 1"

Découper un programme en fonctions permet :

- de réutiliser plusieurs fois le même traitement ;
- d'éviter les répétitions de code ;
- de rendre le programme plus lisible ;
- de tester séparément chaque partie du programme.

Cette manière de programmer s'appelle la **programmation procédurale**. Le principe est de décomposer un problème complexe en sous-problèmes plus simples.

"Exemple 2"

Les paramètres d'une fonction ne sont pas forcément numériques.

Exemple avec un booléen :

```
def negation(val: bool) -> bool:
    """
    Prend en paramètre une valeur booléenne.
    Renvoie sa négation.
    """
    return not val
```

Exemple d'appel : `negation(True)` renvoie `False`.

Exemple avec une chaîne de caractères :

```
def perdu(nom: str) -> str:
    """
    Prend en paramètre un nom de type str.
    Renvoie un message personnalisé de type str.
    """
    return "Vous avez perdu " + nom
```

Exemple d'appel : `perdu("Alice")` renvoie `"Vous avez perdu Alice"` .

"Exercice 1"

1. Écrire une fonction `fahrenheit` qui prend en paramètre une température en degrés Celsius C et renvoie sa valeur F en degrés Fahrenheit.
   On rappelle la formule : $F = 1{,}8 \times C + 32$.

2. Le code de la fonction suivante est-il correct ? Justifier.

```
def victoire(nom: str) -> str:
return "Vous avez gagné " + nom
```

Fonction avec plusieurs paramètres ou plusieurs `return`

"Exercice 2"

1. On considère la fonction suivante :

```
def mystere(x: int, y: int) -> int:
    if x >= y:
        return x
    else:
        return y
```

- Que renvoie `mystere(1, 2)` ? `mystere(1, 1)` ? `mystere(2, 1)` ? `mystere(-1, -2)` ?

- Écrire une spécification de cette fonction qui pourrait être ajoutée dans une docstring.

2. On considère la fonction suivante :

```
def val_abs(x: float) -> float:
    """
    Prend en paramètre un flottant x.
    Renvoie sa valeur absolue.
    """
    return x
    if x < 0:
        return -x
```

- Que renvoie `val_abs(3)` ? Et `val_abs(-3)` ?

- Corriger si besoin le code de cette fonction.

3. On considère la fonction suivante :

```
def mention(note: float) -> str:
    if note < 10:
        return "refusé"
    elif note < 12:
        return "passable"
    elif note < 14:
        return "assez bien"
    elif note < 16:
        return "bien"
    else:
        return "très bien"
```

- Que renvoie `mention(8.5)` ? `mention(11.9)` ? `mention(16.4)` ? `mention(12.9)` ? `mention(14.1)` ?

- Écrire une spécification de cette fonction qui pourrait être ajoutée dans une docstring.

"Propriété 1"

Une fonction peut avoir plusieurs paramètres.

Exemple :

```
def aire_rectangle(longueur: float, largeur: float) -> float:
    """
    Prend en paramètres la longueur et la largeur d'un rectangle.
    Renvoie son aire.
    """
    return longueur * largeur
```

Une fonction peut aussi contenir plusieurs instructions `return`, par exemple dans une structure conditionnelle.

Exemple :

```python
def maximum(x: int, y: int) -> int:
    """
    Prend en paramètres deux entiers x et y.
    Renvoie le plus grand des deux.
    """
    if x >= y:
        return x
    else:
        return y
```

Dès qu'une instruction `return` est exécutée, l'évaluation de la fonction s'arrête immédiatement.

Les instructions situées après ce `return`, dans le même appel de fonction, ne sont pas exécutées.

Fonction sans paramètre ou sans return

"Exercice 3"

1. On considère la fonction suivante :

   ```python
   def mystere2(n):
       for k in range(n):
           print("Bonjour")
   ```

   Que se passe-t-il si on exécute `mystere2(30)` ?

   ---

2. Saisir la fonction suivante dans une console Python :

   ```python
   def epoch() -> str:
       return "1970/01/01"
   ```

   Que renvoie `epoch` ? Et `epoch()` ?

   ---

3. On considère le programme suivant :

```
def carre1(x):
    return x ** 2

def carre2(x):
    print(x ** 2)

a = carre1(2)
b = carre2(2)
print("Type de a ", type(a))
print("Type de b ", type(b))
```

- Recopier et tester ce programme. Quelles sont les valeurs affichées pour les types des variables `a` et `b` ?

- Ajouter des annotations de types et une docstring à chacune des fonctions `carre1` et `carre2` .

"Propriété 2"

Une fonction peut être définie sans paramètre.

Dans ce cas, les parenthèses restent obligatoires dans la définition et dans l'appel.

```
def epoch() -> str:
    """
    Renvoie la date de référence du temps Unix.
    """
    return "1970/01/01"
```

Appel correct : `epoch()` .

Une fonction peut aussi être définie sans instruction `return` .

Dans ce cas, elle renvoie automatiquement la valeur spéciale `None` .

Exemple :

```
def afficher_bonjour() -> None:
    """
    Affiche le message Bonjour.
    Ne renvoie aucune valeur utile.
    """
    print("Bonjour")
```

Il faut bien distinguer `print` et `return` .

- `print` affiche une valeur à l'écran.
- `return` renvoie une valeur au programme qui a appelé la fonction.

Exemple :

```
def cube1(x: int) -> int:
    return x ** 3

def cube2(x: int) -> None:
    print(x ** 3)
```

La fonction `cube1` renvoie une valeur réutilisable dans un calcul.
La fonction `cube2` affiche une valeur mais renvoie `None` .

Modules

"Définition 2"

Un **module Python** est un fichier contenant du code Python, en particulier des fonctions que l'on peut réutiliser dans d'autres programmes.

On peut comparer un module à une bibliothèque de fonctions.

Exemple : le module `math` contient des fonctions mathématiques comme `sqrt` , qui calcule une racine carrée. Pour l'utiliser dans un programme, il faut importer le module `math` .

Syntaxe recommandée pour un import de module :

```
import math
# on préfixe la fonction du nom du module
r2 = math.sqrt(2)
```

Syntaxe possible si on n'utilise qu'une fonction du module :

```
from math import sqrt
r2 = sqrt(2)
```

Syntaxe déconseillée :

```
from math import *
r2 = sqrt(2)
```

Cette forme importe tous les noms du module `math` .

Elle est dangereuse dans un programme long, car elle peut provoquer des conflits entre des noms de fonctions ou de variables.

"Module `turtle` "

Le module `turtle` est une implémentation en Python du langage Logo, créé dans les années 1970 pour l'enseignement de l'informatique. Il est disponible dans la distribution standard de Python.

Le module `turtle` permet de tracer des figures géométriques en déplaçant une pointe de stylo, souvent représentée par une tortue.

La fenêtre graphique utilise un repère cartésien dont l'origine est au centre de la fenêtre. L'unité par défaut est le pixel.

Quand le crayon est baissé, les déplacements laissent une trace. Quand le crayon est levé, les déplacements ne tracent rien.

Nous utiliserons les fonctions suivantes.

Fonction	Spécification
`turtle.goto(x, y)`	déplace la tortue jusqu'au point de coordonnées `(x, y)`

Fonction	Spécification
`turtle.up()`	lève le crayon
`turtle.down()`	baisse le crayon
`turtle.setheading(angle)`	choisit l'angle d'orientation de la tortue, en degrés
`turtle.forward(n)`	avance de `n` pixels selon l'orientation de la tortue
`turtle.left(a)`	tourne à gauche de `a` degrés
`turtle.right(a)`	tourne à droite de `a` degrés
`turtle.color("red")`	choisit la couleur rouge ; on peut aussi utiliser `"black"`, `"green"`, `"blue"`, etc.

On donne ci-dessous une fonction permettant de tracer un polygone régulier de `n` côtés de longueur `30` pixels.

```
import turtle

def polygone(n: int) -> None:
    """
    Prend en paramètre un entier n.
    Trace un polygone régulier à n côtés.
    Ne renvoie aucune valeur utile.
    """
    angle = 360 / n
    for k in range(n):
        turtle.forward(30)
        turtle.left(angle)

# programme principal
# positionnement de la tortue à l'origine
turtle.up()
turtle.goto(0, 0)
turtle.down()
# tracé d'un hexagone
polygone(6)
# boucle d'interaction
turtle.mainloop()
# turtle.done() sur Capytale
```

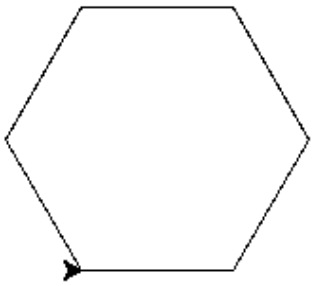
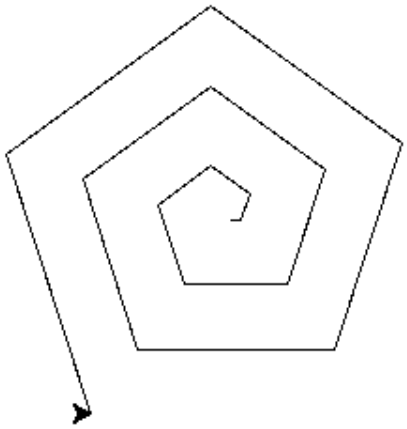
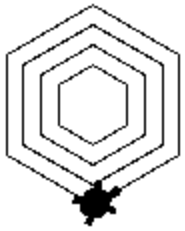
"Exercice 4"

Dans cet exercice on utilisera le module `turtle` .

1. Écrire une fonction `spirale1(n)` qui permet de tracer une spirale constituée de `n` carrés déformés.

2. Écrire une fonction `spirale2(n, m)` qui permet de tracer une spirale constituée de `n` polygones déformés à `m` côtés.

3. Écrire une fonction `spirale3(n, m)` qui permet de tracer une spirale constituée de `n` polygones concentriques à `m` côtés.

Polygone régulier	Spirale	Polygones concentriques
		

Portée d'une variable

"Exercice 5"

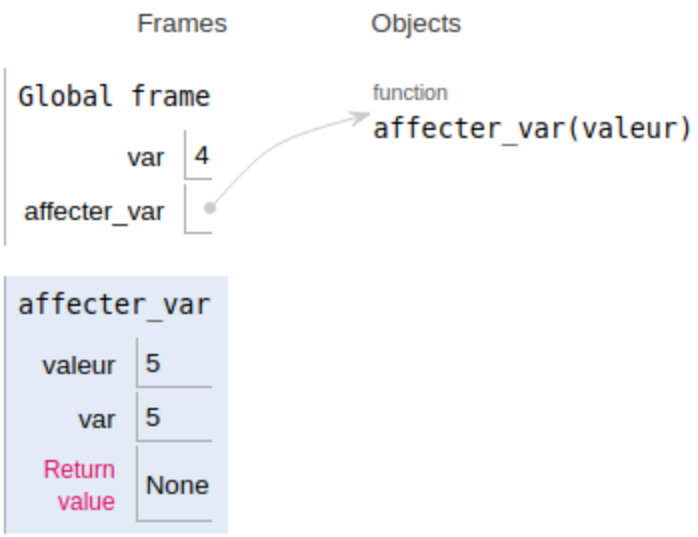
1. Saisir puis exécuter le code ci-dessous dans la zone script d'un IDE Python :

```
var = 4

def affecter_var(valeur):
    var = valeur

print(var)
affecter_var(5)
print(var)
```

2. Le comportement est-il conforme à ce qu'on attendrait ?

---

3. Copier puis tester le code dans Python Tutor.
   Juste après l'évaluation de `affecter_var(5)` , on obtient la visualisation suivante de l'état des variables :



- Combien existe-t-il de variables nommées `var` à cette étape ?

- Que se passe-t-il lorsque l'évaluation de la fonction est terminée et qu'on passe à l'instruction suivante ?

"Définition 3"

La **portée** d'une variable est la partie du programme dans laquelle ce nom de variable est accessible.

Une variable définie dans le programme principal est une variable de **portée globale**.

Une variable définie à l'intérieur d'une fonction est une variable de **portée locale**.

Les paramètres d'une fonction sont aussi des variables locales à cette fonction.

Exemple :

```
def f(x):
    y = x + 1
    return y
```

Dans cette fonction, `x` et `y` sont des variables locales.

Une variable locale n'existe que pendant l'exécution de la fonction. Elle disparaît lorsque l'appel de fonction est terminé.

Si une variable locale porte le même nom qu'une variable globale, le nom local est prioritaire à l'intérieur de la fonction.
On dit que la variable locale **masque** la variable globale.

"Exercice 6"

On considère le programme suivant :

```
def incrementer(a):
    return a + 1

x = 4
incrementer(x)
x = incrementer(x)
```

Compléter le tableau d'évolution de la variable `x` .

Instruction exécutée	Valeur de `x`
avant la ligne 4	non définie
ligne 4 : `x = 4`	_____
ligne 5 : `incrementer(x)`	_____
ligne 6 : `x = incrementer(x)`	_____

"Erreur : `UnboundLocalError` "

On considère le programme suivant :

```
var = 4

def incrementer_var():
    var = var + 1

incrementer_var()
```

L'exécution de `incrementer_var()` produit l'erreur suivante :

```
UnboundLocalError: local variable 'var' referenced before assignment
```

Dans la fonction `incrementer_var` , l'instruction suivante contient une affectation à la variable `var` : `var = var + 1` .
Python considère donc que `var` est une variable locale à la fonction.
Mais pour calculer `var + 1` , Python doit d'abord connaître l'ancienne valeur de cette variable locale.
Or cette variable locale `var` n'a pas encore reçu de valeur. C'est pourquoi Python produit une erreur `UnboundLocalError` .

Le point important est le suivant : dans une fonction, **le nom local est prioritaire sur le nom global**.

"Remarque 2"

Python possède un mot-clé `global` , qui permet de modifier une variable globale depuis une fonction.

Exemple :

```
vies = 3

def perdre_une_vie():
    global vies
    vies = vies - 1
```

Cette écriture peut sembler pratique, par exemple pour gérer le nombre de vies dans un jeu vidéo.

Cependant, l'utilisation de `global` est généralement une mauvaise pratique dans les programmes longs. Elle rend le programme plus difficile à comprendre, car la modification de la variable n'est visible ni dans ses paramètres ni dans sa valeur de retour.

On préfère souvent écrire une fonction qui reçoit une valeur en paramètre et renvoie une nouvelle valeur. Exemple plus propre :

```
def perdre_une_vie(vies: int) -> int:
    return vies - 1

vies = 3
vies = perdre_une_vie(vies)
```

Mise au point

"Exercice 7"

On considère la fonction ci-dessous qui contient une erreur :

```
def carre(x):
    """
    Prend en paramètre un nombre x.
    Renvoie son carré.
    """
    return x * 2
```

1. Saisir cette fonction dans la zone de script d'un IDE Python puis exécuter le code.
2. Ajouter l'instruction suivante au code, exécuter puis commenter :

   ```
   assert carre(2) == 4, "Erreur sur le carré de 2"
   ```

---

3. Ajouter l'instruction suivante au code, exécuter puis commenter :

   ```
   assert carre(-2) == -4, "Erreur sur le carré de -2"
   ```

---

4. Compléter une fonction de jeu de tests regroupant plusieurs tests en essayant de couvrir les différents cas possibles :

```
def test_carre():
    assert carre(2) == 4, "Erreur sur le carré de 2"
    assert carre(-2) == 4, "Erreur sur le carré de -2"
    # à compléter
    ...
    ...
    ...
    ...
    ...
    ...
```

"Définition 4"

Un **test unitaire** est une instruction qui permet de vérifier si la valeur renvoyée par une fonction sur une entrée fixée est celle attendue.

En Python, on peut écrire un test avec l'instruction `assert` selon la syntaxe suivante :

```
assert fonction(valeur_parametre) == resultat_attendu, "message d'erreur"
```

Le message d'erreur est optionnel.

On peut regrouper plusieurs tests dans une fonction de jeu de tests.

Exemple :

```
def test_carre():
    assert carre(2) == 4
    assert carre(-2) == 4
    assert carre(0) == 0
    print("Tous les tests ont été réussis")
```

Un bon jeu de tests doit essayer de couvrir plusieurs cas : valeurs positives, négatives, nulles et cas limites.