1 Fonctions pures

Afin de bâtir des logiciels complexes, il est essentiel de décomposer notre programme en briques logicielles indépendantes. Les fonctions nous permettent une telle décomposition. Dans sa forme la plus simple, une fonction prend en entrée une valeur et renvoie une valeur calculée à partir de la valeur d'entrée. Par exemple, la fonction

```
>>> def carre(n):
        return n * n
```

prend en entrée la valeur `n` et renvoie la valeur `n * n`. On accède ensuite à la fonction de la manière suivante :

```
>>> carre(3)
 9
```

Bien entendu, il est possible d'utiliser le résultat renvoyé par une fonction à l'intérieur d'une expression.

```
>>> carre(3) + carre(4)
 25
```

Ces fonctions sont dites pures, dans la mesure où elles ne changent pas l'état du système. Attention, ce n'est pas la fonction qui a choisi d'afficher son résultat, mais la boucle interactive qui a appelé la fonction `print` sur le résultat obtenu en évaluant l'expression. Les fonctions pures n'ont aucun effet de bord et le résultat qu'elles renvoient ne dépend pas de l'état du système. Elles sont donc très proches des fonctions mathématiques.

Une fonction peut utiliser des variables dites *locales* pour produire le résultat demandé. Par exemple, la fonction

```
>>> def est_premier(n):
        if n <= 1:
            res = False
        else:
            res = True
            for k in range(2, n):
                if n % k == 0:
                    res = False
        return res
>>> est_premier(19)
 True
```

renvoi `True` si l'entier passé en paramètre est premier et `False` dans le cas contraire. L'algorithme teste tous les nombres $2, 3, \ldots, n-1$ et si un de ces nombres est un diviseur de n, la variable `res` est changée en `False`. La valeur de cette variable est ensuite renvoyée.

L'instruction `return` interrompt le flot d'instructions de la fonction et renvoie donc la valeur indiquée par le premier `return` rencontré. La fonction précédente peut donc se simplifier en :

```
>>> def est_premier(n):
        if n <= 1:
            return False
        for k in range(2, n):
            if n % k == 0:
                return False
        return True
```

Cependant, pour des raisons de lisibilité, il est préférable de ne pas trop multiplier les `return` et donc les points de sortie.

Exercice 1

Soit les fonctions définies en Python par :

```
def f(n):
    res = 0
    for i in range(1, n + 1):
        res = res + 2 * i
    for j in range(1, n + 1):
        res = res + 3 * j
    return(res)
```

```
def g(n):
    res = 0
    for i in range(1, n + 1):
        for j in range(1, n + 1):
            res = res + 2 * i + 3 * j
    return(res)
```

1. Que font ces fonctions ?
2. Exprimer en fonction de n, le nombre d'additions effectuées par chacune de ces deux fonctions.

Exercice 2

On cherche à créer une fonction `binome` qui pour n et p entiers renvoie :

$$\binom{n}{p} = \begin{cases} \dfrac{n!}{p!(n-p)!} & \text{si } 0 \leqslant p \leqslant n \\ 0 & \text{sinon} \end{cases}.$$

1. On note H_i l'hypothèse « `res` contient $(i-1)!$ ». Compléter les fonctions suivantes de sorte que la fonction `binome_naif` ait l'effet recherché :

```
def factorielle(n):
    res = __ # H_1 est  vraie
    for i in range(__):
        # Si H_i est vraie,
        res = __
        # alors H_(i+1) est vraie.
    # H_(n+1) est vraie
    return(res)

def binome_naif(k, n):
    if 0 <= k and k <= n:
        return factorielle(n) / (factorielle(k) * factorielle(n-k))
    else:
        return _
```

2. En étudiant le nombre de multiplications et de divisions effectuées et le type du résultat renvoyé par cette fonction, expliquer ses inconvénients.
3. Pour remédier à ces inconvénients, on peut remarquer que si n et p sont deux entiers tels que $1 \leqslant p \leqslant n$, alors :
$$\binom{n}{p} = \frac{n}{p}\binom{n-1}{p-1}.$$
En déduire une fonction `binome(n, p)` plus efficace que la fonction précédente. On fera en sorte que ce programme utilise moins de $2p$ opérations de multiplication ou division.

---

2 Variables locales et globales

Comme nous avons vu, il est possible de créer des variables à l'intérieur d'une fonction. Ces variables sont *locales* à la fonction et sont détruites une fois sorti de la fonction.

```
>>> def puissance_quatre(n):
        c = n * n
        return c * c
>>> puissance_quatre(2)
 16
>>> c
 name 'c' is not defined
```

Si la variable `c` est définie avant l'appel de la fonction, sa valeur est masquée lors de l'appel et on la retrouve une fois sorti de la fonction.

```
>>> c = 5
>>> puissanceQuatre(2)
 16
>>> c
 5
```

Autrement dit, une fois dans la fonction, juste avant d'exécuter l'instruction `return`, l'état du système est le suivant :

```
# etat local fonction {c: 4, n: 2}
# etat global         {c: 5}
```

Comme on peut le voir, l'état du système est la superposition de deux états. L'état du dessus correspond à l'état à l'intérieur de la fonction. L'état du dessous correspond à l'état au niveau de l'appel de la fonction. Dans la fonction, la variable globale `c` contenant 5 est masquée par la variable locale `c` contenant 4. Une fois l'appel de fonction fini, l'état local à la fonction est supprimé et on retrouve notre état initial.

```
# etat                {c: 5}
```

Ce mécanisme permet à la fonction `puissance_quatre` de n'avoir aucun effet de bord : une fois sorti de la fonction, on retrouve dans `c` sa valeur initiale. Notons au passage que la variable `n` est locale à la fonction. Elle est initialisée avec la valeur passée lors de l'appel de la fonction. Étant donné que la variable est locale, on peut la changer sans craindre d'effet de bord dans l'état global.

```
>>> def puissance_quatre(x):
        x = x * x
        return x * x
>>> x = 2
>>> puissance_quatre(x)
 16
>>> x
 2
```

Il est théoriquement possible d'accéder aux variables globales. C'est cependant une très mauvaise manière de programmer.

```
>>> def f(a):
        return a + b
>>> b = 1
>>> f(2)
 3
```

Lorsque l'on est dans la fonction pour calculer `f(2)`, juste avant le `return`, l'état du système est donné par :

```
# etat local fonction {a: 2}
# etat global                {b: 1}
```

Autrement dit, à l'intérieur d'une fonction, la variable à laquelle on accède est toujours celle du niveau le plus haut possédant une variable de ce nom. À chaque appel de fonction, un nouvel état local s'ajoute à la pile d'appel. Ce niveau disparaît lorsque l'on sort de la fonction.

Attention cependant, il n'est pas possible de redéfinir une variable qui n'est pas locale. Afin de pouvoir redéfinir une telle variable, il convient de la déclarer à l'intérieur de la fonction comme une variable *globale* avec le mot clé `global`.

```
>>> compteur = 0
>>> def carre(n):
        global compteur
        compteur = compteur + 1
        if compteur <= 10:
            return n * n
        else:
            return 0
```

Cette fonction est prête pour l'obsolescence programmée : après 10 appels, elle renverra toujours 0 et cessera donc de fonctionner.

Comme nous l'avons signalé, l'accès en lecture de variables globales est une *très* mauvaise habitude. En effet, le résultat renvoyé par de telles fonctions dépend de l'état du système. Elles ne sont plus pures. Les variables globales auxquelles on accède en écriture à l'intérieur d'une fonction permettent quant à elles de changer l'état du système. Le dernier élève ayant tenté d'utiliser des variables globales a été placé sur orbite. Il tourne encore. À vous de voir.

Enfin, une fonction peut elle-même appeler une autre fonction. On obtient alors un état composé de plusieurs niveaux :

```
>>> def puissance_deux(n):
        res = n * n
        return res
>>> def puissance_quatre(n):
        u = puissance_deux(n)
        res = puissance_deux(u)
        return res
>>> a = 2
>>> puissanceQuatre(a)
 16
```

Lors du calcul de `puissance_deux(u)`, tout en haut de la pile d'exécution, le système est dans l'état suivant :

```
# etat local puissance_deux                    {n: 4, res: 16}
# etat local puissance_quatre        {n: 2, u: 4}
# etat global                {a: 2}
```

3 Arguments multiples

Une fonction peut prendre en entrée des arguments multiples. Par exemple, le calcul du pgcd de deux entiers positifs peut se faire par la fonction :

```
>>> def pgcd(a, b):
        while b != 0:
            c = a % b
            a = b
            b = c
        return a
>>> pgcd(21, 42)
 7
```

Une fonction peut avoir autant d'arguments que l'on le souhaite. Elle peut même n'avoir aucun argument.

```
>>> def greetings():
        return "Bonjour"
```

4 Renvoi de valeurs multiples, tuples

Une fonction ne peut techniquement renvoyer qu'une seule valeur. C'est parfois problèmatique. Supposons par exemple que l'on souhaite écrire une fonction qui nous donne l'heure en fonction du nombre de secondes qui se sont écoulées depuis minuit. On doit pour cela renvoyer trois entiers : h, m et s.

Pour cela, Python propose un nouveau type appelé `tuple`. Il permet de stocker plusieurs valeurs élémentaires dans une valeur « conteneur ». Par exemple, on peut écrire :

```
>>> heure = (18, 59, 37)
```

On peut ensuite accéder aux différentes composantes de notre conteneur avec l'opérateur « `[]` ». Le premier composant est indexé par 0, le second par 1 et le troisième par 2.

```
>>> type(heure)
 tuple
>>> heure[0]
 18
```

Attention cependant, les tuples sont immutables et il n'est pas autorisé à changer une de ses valeurs élémentaires avec une instruction du type « `heure[0] =` ». Les tuples sont si utiles, que l'on omet souvent d'écrire les parenthèses :

```
>>> heure = 18, 59, 37
```

Mais le meilleur est qu'il est possible de les déconstruire et d'effectuer une affectation multiple :

```
>>> h, m, s = heure
# Ce qui est équivalent à
>>> h = heure[0]
    m = heure[1]
    s = heure[2]
```

Nous avons maintenant à notre disposition d'une instruction très utile pour échanger facilement deux variables :

```
>>> a, b = b, a
```

Enfin, sachez qu'un tuple peut contenir des valeurs de types différents.

Pour en revenir à notre problème initial, voici une fonction qui renvoie l'heure à partir du nombre de secondes qui se sont écoulées depuis minuit.

```
>>> def heure_depuis_secondes(n):
        s = n % 60
        n = n // 60
        m = n % 60
        h = n // 60
        return h, m, s
```

4.1 Procédures

Une fonction peut aussi fonctionner par effet de bord. Un exemple classique est une fonction affichant un résultat à l'écran.

```
>>> def affiche_heure(n):
        h, m, s = heure_depuis_secondes(n)
        print("Il est", str(h) + ":" + str(m) + ":" + str(s))
```

Cette fonction affiche l'heure en fonction du nombre de secondes qui se sont écoulées depuis minuit.

De telles fonctions ne sont pas faites pour renvoyer une valeur. Elles ne possèdent pas d'instruction `return`. Elles fonctionnent par effet de bord et on dit que ce sont des procédures. Afin d'intégrer les procédures dans le cadre plus général des fonctions, elles renvoie la valeur `None`. Cette valeur est l'unique valeur du type `NoneType`.

Enfin, il est possible pour une fonction de produire des effets de bord et de renvoyer une valeur. Encore une fois, c'est une manière de programmer que nous déconseillons vivement. Sauf cas exceptionnel, comme la fonction `input`, cette manière de programmer n'est pas recommandée.

4.2 Les fonctions comme valeur

Les fonctions sont des valeurs comme les autres. Leur type est `function`.

```
>>> def next(n):
        return n + 1
>>> type(next)
 function
```

En particulier, il est possible de passer une fonction en argument à une fonction.

Exercice 3

---

1. On considère une suite (u_n) définie par son premier terme $u_0 = a$ et la relation de récurrence
$$\forall n \in \mathbb{N} \quad u_{n+1} = f(u_n)$$
Écrire une fonction `u(a, n, f)` qui calcule et renvoie u_n.
2. On considère une suite (v_n) définie par ses premiers termes $u_0 = a$, $u_1 = b$, $u_2 = c$ et la relation de récurrence
$$\forall n \in \mathbb{N} \quad u_{n+3} = f(u_n, u_{n+1}, u_{n+2})$$
Écrire une fonction `u(a, b, c, n, f)` qui calcule et renvoie u_n. Combien cette fonction utilise-t-elle d'appels de f ?

Exercice 4

On suppose que f est une fonction codée en Python par la fonction `f`.

1. Écrire une fonction `somme(f, n, m)` qui calcule $\sum_{k=n}^{m} f(k)$.
2. Écrire une fonction `produit(f, n, m)` qui calcule $\prod_{k=n}^{m} f(k)$.

Voici enfin une liste d'exercices sur les fonctions :

Exercice 5

Quel est l'effet des fonctions suivantes ?

```
def f(n):
    for i in range(1, n + 1):
        print(i * i)
```

```
def g(n):
    res=0
    for i range(1, n + 1):
        res = res + i
    return res
```

```
def h(n):
    res=1
    for i range(1, n + 1):
        res = res * i
    return res
```

```python
def h(n, x):
    res = 1
    for i range(n - 1):
        res = res * x
    return res
```

```python
def s(n, valdeb):
    res = valdeb
    for i in range(1, n + 1)
        res = 3 * res**2 + 1
    return res
```

```python
def calculsuite(n, valdeb, f):
    res = valdeb
    for i in range(1, n + 1):
        res = f(res)
    return res
def g(x):
    return(3 * x * x + 1)
calculsuite(4, 0, g)
```