TP3

Partie 1 — Appel de fonction prédéfinies

Ex. 1 — Fonction print

Un des concepts les plus importants en programmation est le concept de fonction. Vous avez déjà manipulé des fonctions dans les précédent TP comme la fonction `print`.

Dans cet exercice, vous allez voir comment appeler une fonction. Une fonction s'appelle en donnant des valeurs que l'on appelle **arguments** entre parenthèses après le nom de la fonction. Si l'appel de fonction a plusieurs arguments, une virgule sépare deux arguments consécutifs.

Un appel de fonction s'écrit donc de la manière suivante :

```
nom_de_la_fonction(argument1, argument2, ...)
```

Les arguments servent à spécifier le comportement de l'appel de la fonction. Par exemple, dans le cas de `print`, les arguments seront les valeurs affichées par la fonction `print`. Les arguments étant des valeurs cela peut correspondre dans le code à :

- des littéraux c'est-à-dire des constantes. Exemples : `15`, `-199`, `12.9`, `"toto"`, `True`, …
- des variables et la valeur ser égale au contenu de la variable. Exemples : `x`, `sum`, …
- un autre appel de fonction. Exemples : `random()`, `sqrt(12)`
- des expressions, c'est-à-dire des valeurs liées par des opérateurs. Exemples : `12 + sum`, `f(12) * 234`, `12 < 24`, `2 * (x+2)`…

Par exemple, pour appeler la fonction `print` pour afficher l'entier `15` on écrit `print(15)`.

La fonction `print` peut être appelée avec un nombre d'arguments quelconque. Les arguments seront affichés dans l'ordre avec un espace les séparant et un saut de ligne à la fin. Un appel `print("La valeur est", value)` affichera donc `La valeur est 100` si `value` est une variable initialisée à `100`.

Vous pouvez remplacer le séparateur par défaut (une chaîne de caractères contenant un espace) par une autre chaîne de caractères (même une chaîne vide), grâce à l'argument `sep`. Il faut pour cela, spécifier la valeur de `sep` en écrivant `sep=valeur` entre les parenthèses de l'appel de fonction.

Exemples :

- `print("Bonjour", "tout", "le", "monde", sep ="*")` affichera `Bonjour*tout*le*monde`

- `print("Bonjour", "tout", "le", "monde", sep ="")` affichera `Bonjourtoutlemonde`

En plus, des arguments qui seront affichés par la focntion `print`, il est possible de changer la chaîne de caractère entre l'affichage de chaque argument (un espace par défaut) ainsi que la chaîne de caractère (un saut de ligne par défaut).

Question 1 : Appeler la fonction `print` pour qu'elle affiche `Hello World!`

Question 2 : Initialiser la variable `your_name` pour qu'elle contienne un chaîne de caractère correpondant à votre nom.

Question 3 : Appeler la fonction `print` pour qu'elle affiche `Hello` suivi d'un espace suivi du contenu de la variable `your_name`.

Question 4 : Utiliser la fonction `print` pour afficher 100 fois le mot `to` sans espace entre les mots.

Correction

```
#Q1
print("Hello World!")

#Q2
your_name = "Arnaud"

#Q3
print("Hello", your_name)

#Q4
for index in range(100):
    print("to", end="")
```

Ex. 2 — Fonction sqrt

En plus des fonctions comme `print` qui effectuent des actions comme afficher du texte, ils existent aussi des fonctions qui calculent des valeurs et **renvoient** (ou **retourne**) donc un résultat.

Un exemple d'une telle fonction est la fonction permettant de calculer la racine carrée d'une valeur. En Python (comme dans beaucoup d'autres langages), cette fonction se nomme `sqrt` qui est une abbriéviation de **square root** qui signifie en anglais la racine carrée.

Un appel de `sqrt(2)` renverra donc la valeur de la racine carrée de 2 qui est environ égal à `1.4142135623730951`.

Pour récupérer la valeur produite par l'appel d'une fonction, il est possible de la stocker dans une variable. Par exemple en écrivant `sqrt2 = sqrt(2)`, on affecte (ou assigne) la valeur (en fait une approximation) de la racine carrée de 2 à la variable `sqrt2`.

On peut aussi utiliser un appel de fonction dans une expression. Par exemple en écrivant `golden_ratio = (1 + sqrt(5))/2`, on affecte la valeur du nombre d'or à la variable `golden_ratio`.

La fonction `sqrt` n'est pas incluse de base dans Python contrairement à la fonction `print`. Elle est dans un module appelé `math` et il faut donc l'importer pour qu'elle soit disponible. La syntaxe (manière d'écrire) pour importer une fonction d'un module est la suivante :

```
from nom_du_module import nom_de_la_fonction
```

Pour importer la fonction `sqrt` du module `math` on a donc ajouter la ligne suivante dans le code :

```
from math import sqrt
```

Il est possible d'importer toutes les fonctions d'un module en mettant `*` en tant que nom de fonction. Par exemple, le code suivant permet d'importer toutes les fonctions du module `math`.

```
from math import *
```

Question 1 : Utiliser la fonction `sqrt` pour initialiser une variable `square_root_of_three` à la valeur de la racine carrée de trois.

Question 2 : Utiliser la fonction `sqrt` pour calculer la distance euclidéenne entre le point de coordonnées (`x1`, `y1`) et le point de coordonnées (`x2`, `y2`) qui représentent deux point dans le plan. Si vous ne connaissez pas la formule permettant de calculer la distance entre deux points, il vous suffit de la rechercher sur internet par exemple au lien suivant.

Correction

```
from math import sqrt

#Q1
square_root_of_three = sqrt(3)

x1 = 0
y1 = 0
```

```
x2 = 1
y2 = 1

#Q2
distance = sqrt((x1 - x2)**2 + (y1 - y2)**2)
```

Partie 2 — Définitions de procédures

Ex. 3 — Définitions de procédures sans paramètres

On vient de voir comment utiliser des fonctions qui existe déjà en Python. Vous allez maintenant définir vos propres fonctions.

La syntaxe pour la définition d'une fonction sans paramètre et donc sans valeur à passer en argument lors d'un appel (on verra dans l'exercice comment rajouter des paramètres) est la suivante :

```
def nom_de_la_fonction():
    ...
    bloc instructions
    ...
```

On peut faire quelques remarques :

- Le mot-clé `def` s'utilise de manière relativement similaire à `if` et `while`. La ligne doit se terminer par un double point suivi d'un bloc d'instructions qui doit être indenté.
- Tout comme pour les noms de variables, vous avez une liberté quasi-totale pour le nom des fonctions. Il vous faut néanmoins respecter les mêmes règles : pas d'espaces (on utilise `_` pour séparer les mots si le nom de la fonction en contient plusieurs), que des minuscules, pas de caractères spéciaux (accents, tilde, parenthèses, accolades, …)

Question 1 : Définir une fonction `print_1000_hello` qui affiche 1000 fois "Hello!" avec un saut à la ligne entre chaque affichage.

Question 2 : Appeler la fonction `print_1000_hello` deux fois.

Question 3 : Définir une fonction `print_1_to_10000` qui affiche les entiers de 1 à 10000 avec un saut à la ligne entre chaque affichage.

Question 4 : Appeler la fonction `print_1_to_10000` trois fois.

Correction

```
#Q1
def print_1000_hello():
    for i in range(1000):
        print("Hello!")


#Q2
for i in range(2):
    print_1000_hello()


#Q3
def print_1_to_10000():
    for i in range(1,10001):
        print(i)


#Q4
for i in range(3):
    print_1_to_10000()
```

Ex. 4 — Définitions de procédures avec paramètres

La syntaxe pour la définition d'une fonction ayant des paramètre est la suivante :

```
def nom_de_la_fonction(nom_parametre1, nom_parametre2, ...):
    ...
    bloc instructions
    ...
```

La liste des paramètres spécifie quelles informations il faudra fournir en guise d'arguments lorsque l'on voudra utiliser cette fonction.

Comme nous l'avons vu à l'exercice précédent les parenthèses peuvent parfaitement rester vides si la fonction ne nécessite pas d'arguments.

Question 1 : Définir une fonction `print_n_hello(n)` qui affiche `n` fois "Hello!" avec un saut à la ligne entre chaque affichage.

Question 2 : Appeler la fonction `print_n_hello(n)` avec comme argument 10.

Question 3 : Écrire une fonction `print_line_multiplication_table(line_number)` qui affiche la ligne d'une table de multiplication, c'est-à-dire tous les multiples de l'entier `line_number` de une fois `line_number` à dix fois `line_number`. Chaque multiple devra être séparé par un caractère `"\t"` qui correspond à une

tabulation. Par exemple, un appel `print_line_multiplication_table(3)` devra afficher la ligne suivante :

```
3   6   9   12  15  18  21  24  27  30
```

Question 4 : Appeler la fonction `print_line_multiplication_table(line_number)` avec comme argument 3.

Question 5 : Écrire une fonction `print_multiplication_table()` qui affiche les lignes 1 à 10 d'une table de multiplication. Un appel `print_multiplication_table()` devra afficher le texte suivant :

```
1   2   3   4   5   6   7   8   9   10
2   4   6   8   10  12  14  16  18  20
3   6   9   12  15  18  21  24  27  30
4   8   12  16  20  24  28  32  36  40
5   10  15  20  25  30  35  40  45  50
6   12  18  24  30  36  42  48  54  60
7   14  21  28  35  42  49  56  63  70
8   16  24  32  40  48  56  64  72  80
9   18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90  100
```

Question 6 : Appeler la fonction `print_multiplication_table()`

Correction

```
#Q1
def print_n_hello(n):
    for i in range(n):
        print("Hello!")

#Q2
print_n_hello(10)

#Q3
def print_line_multiplication_table(line_number):
    for i in range(1, 11):
        print(i * line_number, end="\t")
```

```
    print()

#Q4
print_line_multiplication_table(3)

#Q5
def print_multiplication_table():
    for i in range(1, 11):
        print_line_multiplication_table(i)

#Q6
print_multiplication_table()
```

Partie 3 — Définitions de fonctions

Ex. 5 — Fonctions ayant une unique instruction retour

Comme nous l'avez vu dans la deuxième question de l'exercice 1, ils existent des fonctions comme `sqrt` calculant et renvoyant (ou retournant) un résultat. Afin d'indiquer à l'ordinateur qu'une fonction doit renvoyer une valeur, il faut utiliser le mot-clé `return` suivi de la valeur que l'on souhaite retourner à l'intérieur du code de la fonction.

On rappelle que les valeurs peuvent être exprimées par :

- des littéraux c'est-à-dire des constantes. Exemples : `15`, `-199`, `12.9`, `"toto"`, `True`, …
- des variables et la valeur sera égale au contenu de la variable. Exemples : `x`, `s`, …
- un appel de fonction. Exemples : `random()`, `sqrt(12)`
- des expressions, c'est-à-dire des valeurs liées par des opérateurs. Exemples : `12 + s`, `f(12) * 234`, `12 < 24`, `2 * (x+2)`…

Par exemple le code suivant défini une fonction `product` qui calcule et renvoie le produit de deux nombres `number1` et `number2`.

```
def product(number1, number2):
    return number1 * number2
```

Une fois que la fonction `product` est définie, on peut l'appeler avec des arguments et récupérer la valeur calculée. Considérons le code suivant :

```
p1 = product(12, 10)
P2 = product(p1, 2)
print(p2)
```

1. La variable `p1` prend la valeur du produit de `12` fois `10` soit `120`

2. La variable `p2` prend la valeur du produit de la valeur de `p1` (égale à `120`) et de `2` soit `240`.
3. Le contenu de la variable `p2` est affiché et donc 240 est affiché en sortie.

Question 1 : Définir une fonction `addition` ayant deux paramètres `number1` et `number2` et qui renvoie la somme des deux nombres `number1` et `number2`

Question 2 : Utiliser la fonction `addition` pour calculer la somme de 12 et 14 et mettre le résultat dans la variable `s`.

Question 3 : Afficher le contenu de la variable `s`.

Correction

```
#Q1
def addition(number1, number2):
    return number1 + number2

#Q2
s = addition(12,14)

#Q3
print(s)
```

Ex. 6 — Plusieurs instructions de retour

Dans certains cas, il peut y arriver que le code d'une fonction contienne plusieurs occurences du mot-clé `return`.

Considérons la fonction suivante nommée `find_multiple_in_interval` :

```
def find_multiple_in_interval(value, begin, end):
    for i in range(begin, end):
        if i % value == 0:
            return i
    return -1
```

Cette fonction parcoure tous les entiers compris dans l'interval [begin , end[et si elle trouve un multiple de `value` dans l'intervalle, elle le renvoie. Dés que l'instruction `return` est exécutée, on sort de la fonction. S'il y a plusieurs multiples de `value` dans l'intervalle, seul le premier multiple est renvoyé, car le `return` arrête l'exécution de la fonction et le reste des entiers de l'intervalle n'est pas parcouru. Si à la fin de la boucle, aucun des entiers de l'intervalle étaient un multiple de `value` la valeur `-1` est renvoyé

Question 1 : Écrire la fonction `absolute_value` ayant un paramètre `x` et qui renvoie la valeur absolue de `x`.

Question 2 : Afficher la valeur absolue de `-11`.

Correction

```
#Q1
def absolute_value(x):
    if x < 0:
        return -x
    return x


#Q2
print(absolute_value(-11))
```

Ex. 7 — Variables locales

Lorsque nous définissons des variables à l'intérieur du corps d'une fonction, ces variables ne sont accessibles qu'à la fonction elle-même. On dit que ces variables sont des variables locales à la fonction.

Les variables à l'intérieur de la fonctions sont donc différentes de celles à l'extérieur de la fonction même si elles ont le même nom.

Question 1 : Créer une variable nommée `a` et qui a une valeur égale à `10`.

Question 2 : Définir une fonction `set_a_to_12` sans paramètre et qui initialise une variable `a` avec la valeur `12`.

Question 3 : Appeler la fonction `set_a_to_12`.

Question 4 : Afficher la valeur de la variable `a`.

Correction

```
#Q1
a = 10


#Q2
```

```
def set_a_to_12():
    a = 12


#Q3
set_a_to_12()


#Q4
print(a)
```

Ex. 8 — Appel par nom

Lorsqu'une variable est passé comme argument d'un appel de fonction, c'est seulement sa valeur qui est transmise et pas la variable elle-même. Un appel de fonction ne peut donc pas modifié la valeur d'une variable. Vous allez pouvoir vérifier cette affirmation avec cet exercice.

Attention vous verrez par la suite des objets (les liste par exemple) dont l'état sera modifiable y compris par un appel de fonction.

Question 1 : Créer une variable `a` ayant pour valeur `0`.

Question 2 : Définir un fonction `increment` ayant un paramètre `value`. Cette fonction devra augmenter de un la valeur de `value` et renvoyer la nouvelle valeur de `value`.

Question 3 : Appeler la fonction `increment` avec comme argument `a`.

Question 4 : Afficher la valeur de `a`.

Question 5 : Appeler la fonction `increment` avec comme argument `a` et stocker le retour de la focntion dans la variable `a`.

Question 6 : Afficher la valeur de `a`.

Correction

```
#Q1
a = 0


#Q2
def increment(value):
```

```
    value += 1
    return value


#Q3
increment(a)


#Q4
print(a)


#Q5
a = increment(a)


#Q6
print(a)
```

Partie 5 — Complex functions

Ex. 9 — Géométrie

Question 1 : Écrire une fonction `rectangle_area` ayant deux paramètres `height` et `length` et qui renvoie l'aire du rectangle correspondant.

Question 2 : Écrire une fonction `rectangle_perimeter` ayant deux paramètres `height` et `length` et qui renvoie le périmètre du rectangle correspondant.

Question 3 : Écrire une fonction `triangle_area` ayant trois paramètres `side1`, `side2` et `side3` (les longueurs des trois cotés) et qui renvoie l'aire du triangle correspondant. On utilisera pour cela la formule de Héron.

Question 4 : Écrire une fonction `triangle_perimeter` ayant trois paramètres `side1`, `side2` et `side3` (les longueurs des trois cotés) et qui renvoie le périmètre du triangle correspondant.

Question 5 : Écrire une fonction `disk_area` ayant un paramètre `radius` et qui renvoie l'aire du disque correspondant. Pour la valeur de pi, on utilisera `pi` qu'on a importé du module math (deuxième ligne du code).

Question 6 : Écrire une fonction `disk_perimeter` ayant un paramètre `radius` et qui renvoie le périmètre du disque correspondant.

Correction

```
from math import sqrt
from math import pi


#Q1
def rectangle_area(length, width):
    return length * width

#Q2
def rectangle_perimeter(length, width):
    return 2 * (length + width)

#Q3
def triangle_area(side1, side2, side3):
    p = (side1 + side2 + side3)/2
    return sqrt(p * (p-side1) * (p-side2) * (p-side3))

#Q4
def triangle_perimeter(side1, side2, side3):
    return side1 + side2 + side3

#Q5
def disk_area(radius):
    return pi * (radius ** 2)

#Q6
def disk_perimeter(radius):
    return 2*pi*radius
```

Ex. 10 — Suites

Question 1 : Écrire une fonction `sum_from_begin_to_end` ayant deux paramètres `begin` et `end` et qui renvoie la somme des entiers de `begin` à `end-1` compris.

Question 2 : Écrire une fonction `fibonacci` ayant un paramètre `n` et qui renvoie le terme de rang `n` de la suite de Fibonacci.

Question 3 : Écrire une fonction `syracuse_next_term(term)` qui calcule le terme suivant de la suite de Syracuse à partir de la valeur `term` du terme de la suite passée en paramètre. utiliser la division entière (opérateur `//`en Python) afin de n'avoir que des entiers.

Question 4 : Écrire une fonction `print_syracuse_values(initial_term)` qui affiche tous les termes de la suite commençant par le terme initial `initial_term` jusqu'au premier terme égal à 1 compris.

Question 5 : Appeler la fonction `print_syracuse_values` avec comme argument la valeur `15`.

Correction

```
#Q1
def sum_from_begin_to_end(begin, end):
    s = 0
    for i in range(begin, end):
        s += i
    return s

#Q2
def fibonacci(n):
    term = 0
    next_term = 1
    for i in range(n):
        temp = next_term
        next_term += term
        term = temp
    return term

#Q3
def syracuse_next_term(term):
    if term % 2 == 0:
        return term//2
    return 3*term + 1

#Q4
def print_syracuse_values(initial_term):
    term = initial_term
    while term != 1:
        print(term)
        term = syracuse_next_term(term)
    print(term)

#Q5
print_syracuse_values(15)
```