

1A_python_functions

November 27, 2023

1 Python Functions

Une fonction est un bloc de code qui exécute une tâche spécifique. Une fonction prend des entrées, effectue une action sur ces entrées et renvoie une sortie. Le but de la fonction est de toujours faire la même chose, mais sur différentes entrées.

Les fonctions sont utiles pour plusieurs raisons:

- Elles permettent de réutiliser du code (éviter la duplication de code, raccourcir le code)
- Elles permettent de rendre le code plus lisible (en donnant un nom à une action)
- Elles permettent de rendre le code plus facile à maintenir (en cas de bug, il suffit de corriger la fonction, correction à un seul endroit)

1.1 Types de fonctions

Il existe deux types de fonctions en Python:

- Les fonctions prédéfinies (built-in functions, Standard Library Functions), ces fonctions sont incluses dans Python et peuvent être utilisées directement tel que `print()`, `len()`, `range()`, `type()`, etc.
- Les fonctions définies par l'utilisateur (user-defined functions), ces fonctions sont définies par l'utilisateur ce sont des fonctions dites custom, tel que: `def my_function()`:

1.2 Python function Declaration

La déclaration d'une fonction en Python se fait avec le mot clé `def` suivi du nom de la fonction et des parenthèses `()`.

```
def my_function(arguments):  
    # code  
    return output
```

- `def` est le mot clé pour déclarer une fonction
- `my_function` est le nom de la fonction
- `arguments` est la liste des arguments de la fonction, les arguments sont optionnels
- `return` est le mot clé pour retourner une valeur, le `return` est optionnel

1.3 Appel de fonction en python

Pour appeler une fonction, il suffit d'écrire le nom de la fonction suivi des parenthèses `()`.

```
[1]: def greet():
      print("Hello, World!")

      # Call the function
      greet()
```

Hello, World!

2 Python Function Arguments

Les fonctions peuvent prendre des arguments, les arguments sont des valeurs variables qui sont passées à la fonction lors de l'appel de la fonction. Une fonction peut ne pas avoir d'arguments.

2.0.1 Exemple 1: Python function with parameters

```
[11]: # function with two arguments
def add_numbers(num1: int, num2: int) -> None:
    sum = num1 + num2
    print("Sum: ",sum)

    # function call with two values
    add_numbers(5, 4)

    # On peut aussi utiliser les arguments en les nommant
    add_numbers(num2=4, num1=5)
```

Sum: 9

Sum: 9

2.1 Arguments avec valeurs par défaut

Les arguments peuvent avoir des valeurs par défaut, dans ce cas, si l'argument n'est pas passé à la fonction, la valeur par défaut sera utilisée.

```
[13]: def add_numbers(a = 7, b = 8):
      sum = a + b
      print('Sum:', sum)

      # function call with two arguments
      add_numbers(7, 8)

      # function call with one argument
      add_numbers(7)
      add_numbers(a = 7)

      # function call with no arguments
      add_numbers()
```

```
Sum: 15
Sum: 15
Sum: 15
Sum: 15
```

2.2 Python Function avec un nombre variable d'arguments

Parfois, on ne sait pas à l'avance combien d'arguments seront passés à la fonction. Pour cela, il est possible de définir une fonction avec un nombre variable d'arguments. Pour cela, il faut utiliser le symbole `*` devant le nom de l'argument. Le nombre variable d'arguments, comme son nom l'indique, permet de passer un nombre arbitraire d'arguments à la fonction.

```
[16]: # program to find sum of multiple numbers
```

```
def find_sum(*numbers):
    print(type(numbers))
    result = 0

    for num in numbers:
        result = result + num

    print("Sum = ", result)

# function call with 3 arguments
find_sum(1, 2, 3, 6, 7,8,5,3,2,1,1)

# function call with 2 arguments
find_sum(4, 9)
```

```
<class 'tuple'>
Sum = 39
<class 'tuple'>
Sum = 13
```

Note: Le nombre variable d'arguments est récupéré sous forme de `tuple`, on peut donc utiliser les méthodes des tuples et surtout boucler sur les arguments.

2.3 The return statement

Une fonction peut retourner une valeur avec le mot clé `return`. Si la fonction ne retourne rien, elle retourne `None`.

```
def add_numbers(num1, num2):
    sum = num1 + num2
    return sum
```

Note: L'instruction `return` termine l'exécution de la fonction et retourne une valeur. Tout code après le `return` ne sera pas exécuté.

```
[17]: # function definition
def find_square(num):
    result = num * num
    return result

# function call
square = find_square(3) # <==> square = 9

print('Square:', square)
```

Square: 9

```
[18]: # function that adds two numbers
def add_numbers(num1, num2):
    sum = num1 + num2
    return sum, num1 - num2

# calling function with two values
s, d = add_numbers(5, 4)

print('Sum: ', s, " Diff: ", d)
```

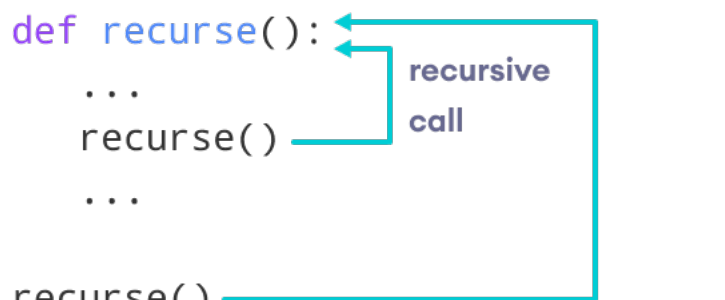
Sum: 9 Diff: 1

2.4 Python Fonctions Récursives

Une fonction récursive est une fonction qui s'appelle elle-même.

Dans le monde réel, on peut observer ce phénomène avec des miroirs, si on place deux miroirs l'un en face de l'autre et un objet entre, il sera réfléchi de façon récursive.

Nous savons qu'en Python une fonction peut en appeler une autre, mais il est également possible qu'une fonction s'appelle elle-même. On appelle ces fonctions des fonctions récursives.



```
def recurse():
    ...
    recurse()
    ...

recurse()
```

Source

Voici un exemple de fonction récursive, la fonction **factorial(x)** calcule la factorielle d'un nombre.

```
[2]: def factorial(n):
    """This is a recursive function
    to find the factorial of an integer"""
```

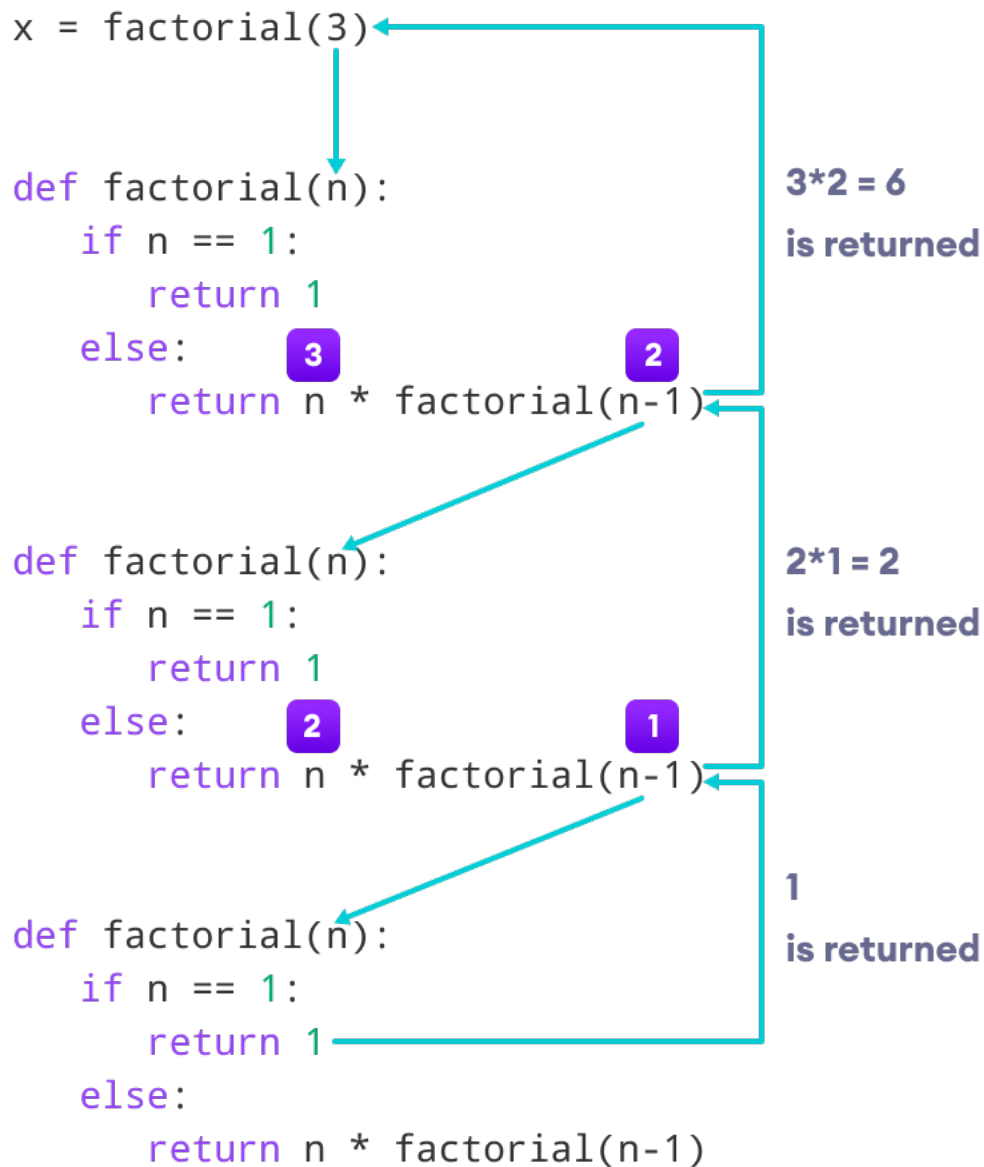
```
if n == 1:
    return 1

print(f"fact({n}) is {n} * fact({n-1})")
return n * factorial(n-1)

factorial(5)
```

5

```
factorial(3)          # 1er appel avec 3
3 * factorial(2)      # 2e appel avec 2
3 * 2 * factorial(1)  # 3e appel avec 1
3 * 2 * 1             # return du 3e appel avec ret=1
3 * 2                 # return du 2e appel
6                     # return du 1er appel
```



[Source](#)

La recursion est terminée lorsque la condition d'arrêt est atteinte, dans notre exemple, la condition d'arrêt est `if num == 1:`.

Il faut toujours avoir un if qui termine la recursion ce if est appelé la condition d'arrêt ou le base case.

Python limite le nombre de recursions à 1000 par défaut, si on dépasse ce nombre, une erreur `RecursionError` est levée. Cette limite peut être modifiée avec la fonction `sys.setrecursionlimit()`. Cette limite est présente pour éviter de consommer trop de mémoire.

```
[3]: def recursor():
      recursor()
      recursor()
```

```
-----
RecursionError                                Traceback (most recent call last)
Cell In[3], line 3
      1 def recursor():
      2     recursor()
----> 3 recursor()

Cell In[3], line 2, in recursor()
      1 def recursor():
----> 2     recursor()

Cell In[3], line 2, in recursor()
      1 def recursor():
----> 2     recursor()

[... skipping similar frames: recursor at line 2 (2970 times)]

Cell In[3], line 2, in recursor()
      1 def recursor():
----> 2     recursor()

RecursionError: maximum recursion depth exceeded
```

2.4.1 Avantages de la recursion

- La recursion permet de résoudre des problèmes complexes en les divisant en sous-problèmes plus simples.
- La recursion permet de résoudre des problèmes qui ne peuvent pas être résolus facilement avec une boucle.
- Les fonctions récursives rendent le code plus lisible et plus élégant.
- Certains problèmes sont plus intuitifs à résoudre avec la recursion. Comme la factorielle, les tours de Hanoi, suite de Fibonacci, etc.

2.4.2 Désavantages de la recursion

- Souvent la logique de la recursion n'est pas évidente.
- Les appels récursifs peuvent être plus coûteux, car ils utilisent la mémoire différemment et ont besoin de `context switching`.
- Les fonctions récursives peuvent être plus compliquées à debugger.

2.5 Python Fonctions anonymes

Les fonctions anonymes sont des fonctions qui n'ont pas de nom. Elles sont définies avec le mot clé `lambda` suivi des arguments et du corps de la fonction.

`lambda argument(s): expression`

Ici, - `lambda` est le mot clé pour définir une fonction anonyme - `argument(s)` est la liste des arguments de la fonction - `expression` est le corps de la fonction

```
[4]: greet = lambda : print('Hello World')

# call the lambda
greet()
```

Hello World

Cette fonction `lambda` ne prend pas d'arguments et affiche simplement Hello World.

2.6 Python Fonctions Lambda avec arguments

Les fonctions `lambda` peuvent prendre un ou plusieurs arguments.

```
[9]: def computation_lst(f, lst):
      for i in lst:
          print(f(i))

l = [1,2,3,4,5]

def square(n):
    return n*n
def double(n):
    return 2*n

computation_lst(lambda n: n*n, l)
computation_lst(lambda n: n*2, l)
```

1
4
9
16
25
2
4
6
8
10

2.7 Python Library Functions

Python a un ensemble de fonctions prédéfinies (built-in functions) qui sont incluses dans Python et peuvent être utilisées directement tel que:

- `print()`
- `sqrt()`
- `pow()`

Certaines fonctions sont incluses dans des modules, il faut importer le module pour pouvoir utiliser la fonction. Comme par exemple, `sqrt()` est défini dans le module `math`.

```
[126]: import math

# sqrt computes the square root
square_root = math.sqrt(4)

print("Square Root of 4 is", square_root)

# pow() computes the power
power = pow(2, 3)

print("2 to the power 3 is", power)
```

Square Root of 4 is 2.0

2 to the power 3 is 8

2.8 Benefits of using functions

1. Réutilisation du code

```
[127]: # function definition
def get_square(num):
    return num * num

for i in [1,2,3]:
    # function call
    result = get_square(i)
    print('Square of', i, '=', result)

for i in [-1,-2,-3]:
    # function call
    print('Square of', i, '=', get_square(i))
```

Square of 1 = 1

Square of 2 = 4

Square of 3 = 9

Square of -1 = 1

Square of -2 = 4

Square of -3 = 9

2. Rendre le code plus lisible

```
[134]: # Helpers function
def get_total_price(quantity, price):
    """Calculate the total price."""
    total = quantity * price
    return total

def apply_discount(total, discount):
    """Apply a discount to the total price."""
    discounted_price = total - (total * discount)
    return discounted_price

def calculate_tax(subtotal, tax_rate):
    """Calculate the tax amount."""
    tax = subtotal * tax_rate
    return tax

# main function
def calculate_final_price(quantity, price, discount, tax_rate):
    """Calculate the final price after applying discount and tax."""
    total = get_total_price(quantity, price)
    discounted_price = apply_discount(total, discount)
    tax = calculate_tax(discounted_price, tax_rate)
    final_price = discounted_price + tax
    return final_price

# Main program
quantity = 10
price = 25.99
discount = 0.1
tax_rate = 0.07

final_price = calculate_final_price(quantity, price, discount, tax_rate)
print("Final Price:", final_price)
```

Final Price: 250.28369999999995

```
[142]: # Using lambda function
def calculate_final_price(quantity, price, discount, tax_rate):
    """Calculate the final price after applying discount and tax."""
    get_total_price = lambda q, p: q*p
    apply_discount = lambda t, d : total - (total * discount)
    calculate_tax = lambda s, tr: s * tr
```

```
total = get_total_price(quantity, price)
discounted_price = apply_discount(total, discount)
tax = calculate_tax(discounted_price, tax_rate)
final_price = discounted_price + tax
return final_price

# Main program
quantity = 10
price = 25.99
discount = 0.1
tax_rate = 0.07

final_price = calculate_final_price(quantity, price, discount, tax_rate)
print(f"Final Price: {final_price:.2f}")
```

Final Price: 250.28

[]: