Les fonctions en Python

Les fonctions

Il arrive souvent qu'un morceau de code particulier soit **répété plusieurs fois dans votre programme.** Il est répété soit littéralement, soit avec seulement quelques modifications mineures, consistant en l'utilisation d'autres variables dans le même algorithme. Il arrive également qu'un programmeur ne puisse pas résister à la simplification du travail et commence à cloner de tels morceaux de code à l'aide des opérations de presse-papiers et de copier-coller.

Les fonctions

Nous pouvons maintenant définir la première condition qui peut vous aider à décider quand commencer à écrire vos propres fonctions : **si un fragment particulier du code commence à apparaître à plusieurs endroits, envisagez la possibilité de l'isoler sous la forme d'une fonction** invoquée depuis les points où le code d'origine a été placé auparavant.

Un bon développeur attentif **divise le code** (ou plus précisément : le problème) en morceaux bien isolés et **les code chacun sous la forme d'une fonction**.

Cela simplifie considérablement le travail du programme, car chaque morceau de code peut être codé séparément et testé séparément. Le processus décrit ici est souvent appelé **décomposition**.

D'où viennent les fonctions ?

En général, les fonctions proviennent d'au moins trois endroits :

• à partir de Python lui-même - de nombreuses fonctions (comme `print()`, `input()`) font **partie intégrante de Python** et sont toujours disponibles sans effort supplémentaire de la part du programmeur ; nous appelons ces fonctions des **fonctions intégrées** ;

• à partir des **modules préinstallés** de Python - de nombreuses fonctions, très utiles, mais utilisées beaucoup moins souvent que celles intégrées, sont disponibles dans un certain nombre de modules installés avec Python ; l'utilisation de ces fonctions nécessite quelques étapes supplémentaires de la part du programmeur afin de les rendre entièrement accessibles (nous vous en parlerons dans un moment);

D'où viennent les fonctions ?

• **directement à partir de votre code** - vous pouvez écrire vos propres fonctions, les placer dans votre code et les utiliser librement ;

Votre première fonction

Vous devez le **définir**. Le mot *définir* est significatif ici.

Voici à quoi ressemble la définition de fonction la plus simple :

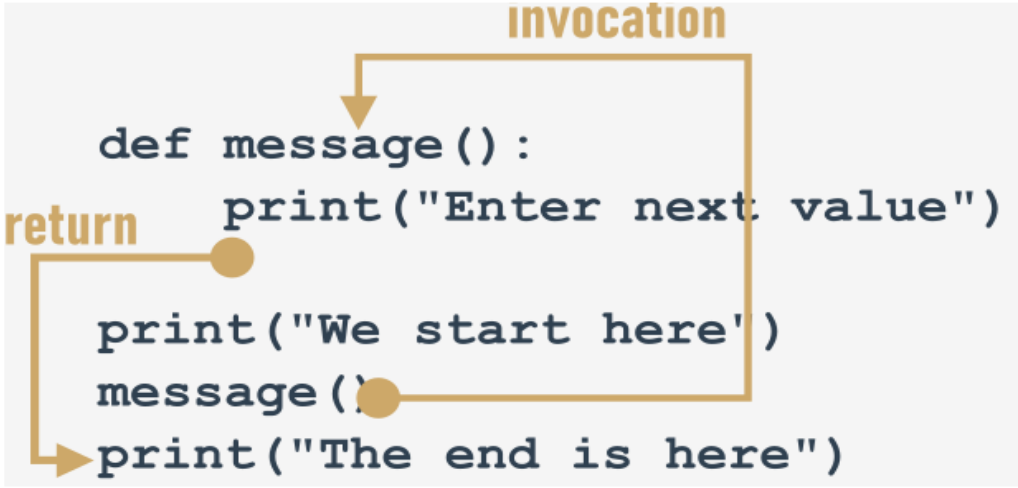
```
def functionName():
    functionBody
```

- Il commence toujours par le mot-clé def (pour définir)
- Ensuite le nom de la fonction.
- Après le nom de la fonction, il y a une place pour une paire de parenthèses (elles ne contiennent rien ici, mais cela changera bientôt)

<u>remarque :</u>

la fonction se termine là où se termine l'imbrication, vous devez donc être prudent.

Fonctionnement des fonctions



Lorsque vous **appelez** une fonction, Python se souvient de l'endroit où cela s'est produit et *saute* dans la fonction invoquée ;

• Le corps de la fonction est alors **exécuté** ;

• Atteindre la fin de la fonction force Python à **retourner** à l'endroit directement après le point d'invocation.

Exemples

```
def nom_fct(n):
    s=0
    for i in range(1,n+1):
        s=s+i
    return s

x=int(input("donner une valeur :"))
som=nom_fct(x)
 print("la somme est ", som)
```

```
def nom_prc(n):
    s=0
    for i in range(1,n+1):
        s=s+i
    print("la somme est ", s)

x=int(input("donner une valeur :"))
nom_prc(x)
```

RESUME

Une **fonction** est un bloc de code qui exécute une tâche spécifique lorsque la fonction est appelée (invoquée). Vous pouvez utiliser des fonctions pour rendre votre code réutilisable, mieux organisé et plus lisible. Les fonctions peuvent avoir des paramètres et des valeurs de retour.

- Vous pouvez définir votre propre fonction à l'aide du mot - clé def
- Vous pouvez définir une fonction qui ne prend aucun argument:

```
def message(): # defining a function
```

RESUME

- Vous pouvez également définir une fonction qui accepte des arguments, tout comme la fonction à un paramètre ci-dessous :

```
def hello(name): # defining a function
```

- Une fonction peut avoir autant de paramètres que vous le souhaitez, mais plus vous en avez, plus il est difficile de mémoriser leurs rôles et leurs objectifs.

Fonctions paramétrées

les paramètres vivent à l'intérieur des fonctions (c'est leur environnement naturel)

- **les arguments existent en dehors des fonctions et sont porteurs de valeurs passées aux paramètres correspondants**.

Fonctions paramétrées

Il est légal et possible d'avoir une variable nommée de la même manière qu'un paramètre de fonction.

L'extrait illustre le phénomène:

```
def message(number):
    print("Enter a number:", number)
number = 1234
message(1)
print(number)
```

Remarque

```
def message(number):
    print("Enter a number:", number)
number = 1234
message(1)
print(number)
```

Fonctions paramétrées

Une situation comme celle-ci active un mécanisme appelé shadowing:

• le paramètre x masque toute variable du même nom, mais ...

• ... uniquement à l'intérieur de la fonction définissant le paramètre.

Le paramètre nommé number est une entité complètement différente de la variable nommée number.

Cela signifie que l'extrait ci-dessus produira la sortie suivante:

Enter a number: 1

1234

Passage de paramètre positionnel

Une technique qui attribue le ième (premier, deuxième, etc.) argument au paramètre de la fonction iième (premier, deuxième et ainsi de suite) est appelée passage de **paramètre positionnel**, tandis que les arguments passés de cette manière sont appelés arguments positionnels.

Vous l'avez déjà utilisé, mais Python peut offrir beaucoup plus. Nous allons vous en parler maintenant.

```
def myFunction(a, b, c):
    print(a, b, c)
myFunction(1, 2, 3)
```

Passage de paramètre positionnel

La fonction suivante sera chargée de présenter quelqu'un:

```
def introduction(firstName, lastName):
    print("Hello, my name is", firstName, lastName)
introduction("Luke", "Skywalker")
```

passage d'argument par mot clé

Python propose une autre convention pour le passage d'arguments, où la signification de l'argument est dictée par son nom, et non par sa position - c'est ce qu'on appelle **le passage d'argument par mot clé**.

```
def introduction(firstName, lastName):
    print("Hello, my name is", firstName, lastName)
introduction(firstName = "James", lastName = "Bond")
introduction(lastName = "Skywalker", firstName = "Luke")
```

passage d'argument par mot clé

- Le concept est clair - les valeurs transmises aux paramètres sont précédées du nom des paramètres cibles, suivi du signe =.
- La position n'a pas d'importance ici - la valeur de chaque argument connaît sa destination sur la base du nom utilisé.

Mélanger les arguments de position et de mot-clé

Vous pouvez mélanger les deux modes si vous le souhaitez - il n'y a qu'une seule règle incassable: vous devez mettre les arguments positionnels avant les arguments mot-clé.

Mélanger les arguments de position et de mot-clé

```
def adding(a, b, c):
    print(a, "+", b, "+", c, "=", a + b + c)
```

• **un pur exemple de passage d'arguments positionnels.**	adding(1, 2, 3)
• **un pur exemple de passage d'arguments mot-clé.**	adding(c = 1, a = 2, b = 3)
• **Regardez l'invocation de fonction.**	adding(3, c = 1, b = 2)

Analysons-le:

l'argument (3) pour le paramètre a est passé en utilisant la manière positionnelle

les arguments pour c et b sont spécifiés comme mots-clés.

Mélanger les arguments de position et de mot-clé

Regardez l'invocation ci-dessous - il semble que nous ayons essayé de définir a deux fois :

adding(3, a = 1, b = 2)

Réponse de Python :

TypeError: adding() got multiple values for argument 'a'

Fonctions paramétrées

Il arrive parfois que les valeurs d'un paramètre particulier soient utilisées plus souvent que d'autres. Ces arguments peuvent avoir leurs valeurs par défaut (prédéfinies) prises en considération lorsque leurs arguments correspondants ont été omis.

La valeur du paramètre par défaut est définie à l'aide d'une syntaxe claire et illustrée :

```
def introduction(firstName, lastName="Smith"):
        print("Hello, my name is", fistName, lastName)
```

introduction("James", "Doe") ➔ Hello, my name is James Doe

introduction("Henry") ➔ Hello, my name is Henry Smith

Remarque

```
def introduction(firstName,
lastName="Smith"):
        print("Hello, my name is", fistName,
lastName)
```

introduction("James", "Doe") ➔ Hello, my name is James Doe

introduction("Henry") ➔ Hello, my name is Henry Smith

Instruction de retour

Return sans expression

Lorsqu'il est utilisé à l'intérieur d'une fonction, il provoque l' arrêt immédiat de l'exécution de la fonction et un retour instantané (d'où le nom) au point d'invocation

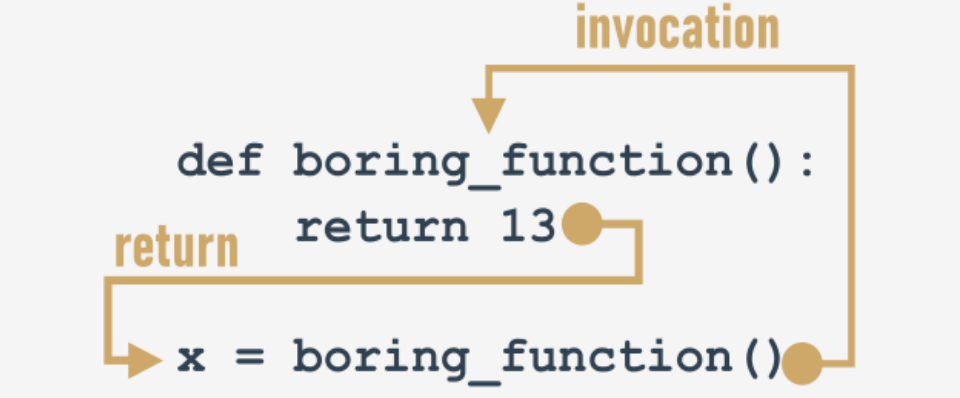
Instruction de retour

Return avec expression

Pour que les **fonctions renvoient une valeur** (mais pas uniquement à cette fin), vous utilisez l' instruction `return`.

- il provoque l' **arrêt immédiat de l'exécution de la fonction** (rien de nouveau par rapport à la première variante)
- de plus, la fonction **évaluera la valeur de l'expression et la renverra (d'où le nom une fois de plus) comme résultat de la fonction** .

Instruction de retour



L'instruction return, enrichie de l'expression (l'expression est très simple ici), "transporte" la valeur de l'expression à l'endroit où la fonction a été invoquée.

Quelques mots sur None

Ses données ne représentent aucune valeur raisonnable - en fait, ce n'est pas du tout une valeur; par conséquent, il ne doit participer à aucune expression .

```
def strangeFunction(n):
        if(n % 2 == 0):
                return True
print(strangeFunction(2)) ➔  true
print(strangeFunction(1)) ➔  false
```

Fonctions et champs d'application

la portée du paramètre d'une fonction est la fonction elle-même. Le paramètre est inaccessible en dehors de la fonction.

```
def scopeTest():
    x = 123
scopeTest()
print(x)
```

Le programme échouera lors de son exécution. Le message d'erreur indiquera:

NameError: name 'x' is not defined

Fonctions et champs d'application

On va vérifier si une variable créée à l'extérieur d'une fonction est visible à l'intérieur des fonctions. En d'autres termes, le nom d'une variable se propage-t-il dans le corps d'une fonction?

```
def myFunction():
    print("Do I know that variable?", var)
var = 1
myFunction()
print(var)
```

Le résultat du test est positif - le code affiche:

```
Do I know that variable? 1
1
```

Remarque

```
def myFunction():
    print("Do I know that variable?", var)
var = 1
myFunction()
print(var)
```

Fonctions et champs d'application

La réponse est: une variable existant en dehors d'une fonction a une portée à l'intérieur des corps des fonctions .

```
def myFunction():
    var = 2
    print("Do I know that variable?", var)
var = 1
myFunction()
print(var)
```

Le résultat a également changé - le code produit maintenant une sortie légèrement différente:

Do I know that variable? 2

1

Remarque

```
def myFunction():
    var = 2
    print("Do I know that variable?", var)
var = 1
myFunction()
print(var)
```

Fonctions et champs d'application

Que s'est il passé?

• la variable créée à l'intérieur de la fonction n'est pas la même que lorsqu'elle est définie à l'extérieur - il semble qu'il y ait deux variables différentes du même nom;

• de plus, la variable de la fonction masque la variable provenant du monde extérieur.

Fonctions et champs d'application

Une variable existant en dehors d'une fonction a une portée à l'intérieur des corps des fonctions, à l'exclusion de celles d'entre elles qui définissent une variable du même nom.

Cela signifie également que la portée d'une variable existante en dehors d'une fonction n'est prise en charge que lors de l'obtention de sa valeur (lecture). L'attribution d'une valeur force la création de la propre variable de la fonction.

le mot-clé global

vous devriez maintenant être parvenu à la question suivante: cela signifie-t-il qu'une fonction n'est pas en mesure de modifier une variable définie en dehors d'elle? Cela créerait beaucoup d'inconfort.

Heureusement, la réponse est non .

Il existe une méthode Python spéciale qui peut étendre la portée d'une variable d'une manière qui inclut les corps des fonctions.

Un tel effet est provoqué par un mot-clé nommé global:

global name

global name1, name2, ...

le mot-clé global

L'utilisation de ce mot-clé dans une fonction avec le nom d'une ou de plusieurs variables, force Python à s'abstenir de créer une nouvelle variable à l'intérieur de la fonction - celle accessible de l'extérieur sera utilisée à la place.

```
def myFunction():
    global var
    var = 2
    print("Do I know that variable?", var)
var = 1
myFunction()
print(var)
```

Le code génère désormais:

Do I know that variable? 2

2

Comment la fonction interagit avec ses arguments

Le code dans l'éditeur devrait vous apprendre quelque chose. Comme vous pouvez le voir, la fonction modifie la valeur de son paramètre. Le changement affecte-t-il l'argument?

```
def myFunction(n):
        print("I got", n)
        n += 1
        print("I have", n)
var = 1
myFunction(var)
print(var)
```

La sortie du code est:

I got 1
I have 2
1

Exemples

a = 1 **def fun():** ** a = 2** ** print(a)** **fun()** **print(a)**	**a = 1** **def fun():** ** global a** ** a = 2** ** print(a)** **fun()** **print(a)**
2 1	2 2

récursivité

Ce terme peut décrire de nombreux concepts différents, mais l'un d'eux est particulièrement intéressant - celui se référant à la programmation informatique.

Dans ce domaine, la récursivité est une technique où une fonction s'invoque .

récursivité

La factorielle a aussi un second côté récursif . Regardez:
n! = 1 × 2 × 3 × ... × n-1 × n
Il est évident que:
1 × 2 × 3 × ... × n-1 = (n-1)!
Donc, finalement, le résultat est:
n! = (n-1)! × n
C'est ici:

```
def factorialFun(n):
    if n < 0:
        return None
    if n < 2:
        return 1
    return n * factorialFun(n - 1)
```

récursivité

La définition des nombres de Fibonacci est un exemple clair de récursivité .
Nous vous avons déjà dit que:
Fib i = Fib i-1 + Fib i-2

```
def fib(n):
    if n < 1:
        return None
    if n < 3:
        return 1
    return fib(n - 1) + fib(n - 2)
```

EXERCICES

- TP 7 :

Les fonctions