**Fonctions**

**Fonctions**

Une fonction est un bloc de code réutilisable qui s'exécute lorsque vous l'appelez. Imaginez functionsune variable qui stocke une fonctionnalité sous la forme d'un bloc de code exécutable plutôt que des données functions.

Les fonctions sont des blocs de code nommés qui sont conçus pour effectuer une tâche spécifique.

Lorsque vous souhaitez effectuer une tâche particulière que vous avez définie dans une fonction, vous appelez le nom de la fonction responsable de celle-ci.

Si vous devez effectuer cette tâche plusieurs fois dans votre programme, vous n'avez pas besoin de saisir tout le code de la même tâche encore et encore ; vous appelez la fonction dédiée à la gestion de cette tâche, et l'appel indique à Python d'exécuter le code à l'intérieur de la fonction.

Les fonctions vous permettent de diviser vos programmes en petites parties, chacune effectuant une tâche spécifique.

En utilisant des fonctions, vous serez en mesure d'écrire du code plus efficace, plus facile à dépanner et à entretenir, et qui peut être réutilisé dans de nombreux programmes différents.

**Syntaxe**

**Exemple d'une fonction simple qui dit bonjour**

def say\_hello():

"""A function that says hello"""

print("Hello!")

say\_hello()

Cet exemple montre la structure la plus simple d'une fonction.

**Définition**

La première ligne utilise le mot-clé defpour informer Python que vous définissez une fonction.

Il s'agit de la définition de la fonction, qui indique à Python le *nom* de la fonction.

Les parenthèses contiennent des informations sur les paramètres ; nous le verrons bientôt. Dans ce cas, le nom de la fonction est say\_hello(), et elle n'a besoin d'aucun paramètre ; ses parenthèses sont donc vides (elles sont néanmoins obligatoires).

Enfin, la définition se termine par deux points et tout le code inclus dans la fonction doit être *indenté* .

**Code**

Toutes les lignes indentées qui suivent def say\_hello():constituent le corps de la fonction.

Le texte de la deuxième ligne est un commentaire appelé *docstring* , qui décrit le rôle de la fonction. *Les docstrings* sont entourés de guillemets triples, que Python recherche lors de la génération de la documentation des fonctions de vos programmes.

La ligne print("Hello!")est la seule ligne de code réel dans le corps de cette fonction, elle say\_hello()n'a donc qu'une seule tâche : imprimer « Bonjour !

**1. Appel de fonction**

Lorsque vous souhaitez utiliser cette fonction, vous l' *appelez* .

Un *appel* de fonction indique à Python d'exécuter le code de la fonction. Pour *appeler* une fonction, saisissez son nom, suivi des paramètres nécessaires entre parenthèses, comme indiqué sur la dernière ligne. Comme aucune information n'est requise ici, l'appel de notre fonction est aussi simple que de saisir say\_hello(). Comme prévu, elle affiche Hello!.

**2. Passer des informations à une fonction**

Une fonctionnalité puissante des fonctions est *les arguments* , ou *paramètres* .

Certaines valeurs des fonctions peuvent être modifiées à chaque appel. Par exemple, la say\_hellofonction doit dire bonjour à quelqu'un, comme « Salut Rick ! », mais comme nous voulons pouvoir modifier ce nom (et également afficher bonjour à « Morty »), nous ne le codons pas en dur dans la fonction.

Au lieu de cela, nous déclarons un *paramètre* username et nous lui passons un *argument .*

def say\_hello(username):

print("Hello "+username)

say\_hello("Rick") # "Rick" is an argument

# output "Hello Rick"

say\_hello("Morty") # "Morty" is an argument

# output "Hello Morty"

* Dans la **définition** de la fonction , nous ajoutons le **paramètre** qui acceptera une valeur.

def say\_hello(username):

print("Hello "+username)

* Dans l' **appel** de fonction , nous passons la valeur (c'est-à-dire un argument).

say\_hello("Rick") # "Rick" is an argument

# output "Hello Rick"

Ici, la fonction exécutera le même code mais avec une valeur différente pour username.

say\_hello("Rick") # "Rick" is an argument

# output "Hello Rick"

say\_hello("Morty") # "Morty" is an argument

# output "Hello Morty"

**Attention !** Si la fonction attend un argument, vous ne pouvez passer qu'un seul argument ; moins ou plus renverra une erreur.

**Exemple d'une fonction qui accepte plus d'un argument**

def say\_hello(username, language):

if language == "EN":

print("Hello "+username)

elif language == "FR":

print("Bonjour "+username)

else:

print("This language is not supported: " + language)

say\_hello("Rick", "FR")

Cette fonction accepte deux arguments :

* Le premier, username, est utilisé pour stocker la valeur du nom d'utilisateur que nous devons saluer.
* Le deuxième, language, est utilisé pour stocker la langue du message d'accueil.

Lors de *l'appel de* la fonction, nous passons deux arguments dans le bon ordre (d'abord le usernamepuis le language).

Les arguments mis en correspondance de cette manière (par ordre) sont appelés *arguments positionnels.*

**Attention :** vous pouvez obtenir des résultats inattendus si vous mélangez l’ordre des arguments dans un appel de fonction lorsque vous utilisez des arguments positionnels.

Si vous obtenez des résultats étranges, vérifiez que l'ordre des arguments dans votre appel de fonction correspond à l'ordre des paramètres dans la définition de la fonction.

**3. Arguments de mots-clés**

Un *argument mot-clé* est une paire nom-valeur que vous passez à une fonction. Vous associez directement le nom et la valeur dans l'argument, évitant ainsi toute confusion lors du passage de l'argument à la fonction.

Les arguments de mots clés vous évitent de vous soucier de l'ordre correct de vos arguments dans l'appel de fonction et ils clarifient le rôle de chaque valeur dans l'appel de fonction.

Réécrivons l’ say\_helloappel de fonction en utilisant *des arguments de mot-clé* .

def say\_hello(username, language):

if language == "EN":

print("Hello "+username)

elif language == "FR":

print("Bonjour "+username)

else:

print("This language is not supported: " + language)

say\_hello(username="Rick", language="FR")

**Attention :** lorsque vous utilisez des arguments de mots-clés, assurez-vous d'utiliser les noms exacts des paramètres dans la définition de la fonction

La fonction say\_hellon'a pas changé. Cependant, lorsque nous l'appelons, nous indiquons explicitement à Python le paramètre auquel chaque argument doit correspondre.

Lorsque Python lit l'appel de fonction, il sait stocker l'argument « Rick » dans le paramètre nom d'utilisateur et l'argument « FR » dans la langue.

L'ordre des arguments des mots-clés importe peu, car Python sait où placer chaque valeur. Les deux appels de fonction suivants sont équivalents :

say\_hello(username="Rick", language="FR")

say\_hello(language="FR", username="Rick")

Vous pouvez utiliser à la fois des arguments *de mot-clé* et *de position* dans le même appel de fonction, mais vous devez placer chaque argument *de position* avant ceux *de mot-clé* .

Par exemple, cela fonctionnera :

say\_hello("Rick", language="FR")

Mais cela ne fonctionnera pas :

say\_hello(username="Rick", "FR")

**4. Valeurs par défaut**

Lors de l'écriture d'une fonction, vous pouvez définir une valeur par défaut pour chaque paramètre.

Si un argument est fourni pour un paramètre lors de l'appel de fonction, Python utilise la valeur de l'argument. Dans le cas contraire, il utilise la valeur par défaut du paramètre.

Ainsi, lorsque vous définissez une valeur par défaut pour un paramètre, vous pouvez exclure l'argument correspondant que vous écririez habituellement dans l'appel de fonction.

L'utilisation de valeurs par défaut peut simplifier vos appels de fonctions et clarifier la manière dont vos fonctions sont généralement utilisées.

Par exemple, si vous constatez que la plupart des appels à say\_hello()sont utilisés pour des salutations en anglais, vous pouvez définir la valeur par défaut de languageà « EN ». Désormais, toute personne appelant say\_helloen anglais peut omettre cette information :

def say\_hello(username, language="EN"):

if language == "EN":

print("Hello "+username)

elif language == "FR":

print("Bonjour "+username)

else:

print("This language is not supported: " + language)

say\_hello("Rick")

# OR

say\_hello(username="Rick")

Nous avons modifié la définition de say\_hellopour inclure une valeur par défaut, « EN », pour language.

Lorsque la fonction est appelée sans languagespécification, Python sait utiliser la valeur « EN » pour ce paramètre.

Pour utiliser une langue autre que l'anglais, vous pouvez utiliser un appel de fonction comme celui-ci :

say\_hello("Rick", "FR")

# OR

say\_hello(username="Rick", language="FR")

Étant donné qu'un argument explicite pour languageest fourni, Python ignorera la valeur par défaut du paramètre.

Lorsque vous utilisez des valeurs par défaut, tout paramètre doté d'une valeur par défaut doit être listé après tous les paramètres qui n'en ont pas. Cela permet à Python de continuer à interpréter correctement les arguments positionnels.

**5. Éviter les erreurs d'argumentation**

Lorsque vous commencez à utiliser des fonctions, ne soyez pas surpris de rencontrer des erreurs liées à des arguments non appariés. Ces erreurs surviennent lorsque vous fournissez moins ou plus d'arguments que nécessaire à la fonction.

Par exemple, voici ce qui se passe si nous essayons d’appeler say\_hello()sans arguments :

Traceback (most recent call last):

File "app.py", line 6, in <module> say\_hello()

TypeError: say\_hello() missing 2 required positional arguments: 'username' and 'language'

La première ligne du traceback nous indique l'emplacement du problème, nous permettant de regarder en arrière et de voir que quelque chose s'est mal passé dans notre appel de fonction.

Dans le deuxième cas, l'appel de fonction incriminé est écrit pour que nous puissions le voir.

Ensuite, lors du troisième, le traceback nous indique que l'appel manque de deux arguments et indique les noms des arguments manquants.

Python est utile car il lit le code de la fonction pour nous et nous indique les noms des arguments à fournir. C'est un autre avantage de donner à vos variables et fonctions des noms descriptifs. Ainsi, les messages d'erreur de Python seront plus utiles à vous et à toute personne susceptible d'utiliser votre code.

Si vous fournissez trop d'arguments, vous devriez obtenir un traceback similaire qui peut vous aider à faire correspondre correctement votre appel de fonction à la définition de fonction.

**6. Portée locale et mondiale**

**Portée locale**

def number\_by\_three(name, day):

sentence = f'Hello {name}! Today is {day}.'

print(sentence)

print(day)

>> NameError: name 'day' is not defined

**Portée mondiale**

name = 'Avner'

def say\_hi():

print(name)

say\_hi()

>> 'Avner'

**7. Valeurs de retour**

Une fonction n'a pas toujours besoin d'afficher directement sa sortie. Elle peut traiter des données puis renvoyer une valeur ou un ensemble de valeurs.

La valeur renvoyée par la fonction est appelée valeur de retour.

L'instruction de retour prend une valeur à l'intérieur d'une fonction et la renvoie à la ligne qui a appelé la fonction.

Les valeurs de retour vous permettent de déplacer une grande partie du travail de base de votre programme dans des fonctions, ce qui peut simplifier le corps de votre programme.

**Renvoyer une valeur simple**

Regardons une fonction qui prend un prénom et un nom et renvoie un nom complet soigneusement formaté :

def get\_formatted\_name(first\_name, last\_name):

"""Return a full name, neatly formatted."""

full\_name = first\_name + ' ' + last\_name

return full\_name.title()

musician = get\_formatted\_name('jimi', 'hendrix')

print(musician)

Jimi Hendrix

La définition de get\_formatted\_name()prend comme paramètres un prénom et un nom.

La fonction combine ces deux noms, ajoute un espace entre eux et stocke le résultat dans full\_name.

La valeur de full\_nameest convertie en titlecas puis renvoyée à la ligne appelante.

Lorsque vous appelez une fonction qui renvoie une valeur, vous devez fournir une variable dans laquelle la valeur de retour peut être stockée.

Dans ce cas, la valeur renvoyée est stockée dans la variable musician. Le résultat affiche un nom proprement formaté, composé des différentes parties du nom d'une personne :

**Exemple** :

def divide\_by\_three(number):

return number / 3

first\_number = 12

first\_number\_computed = divide\_by\_three(first\_number)

print(first\_number\_computed)

>> 4.0

second\_number = 27

second\_number\_computed = divide\_by\_three(second\_number)

print(second\_number\_computed)

>> 9.0

**Renvoyer plusieurs valeurs avec un tuple**

Nous avons déjà vu qu’un tuple peut être déballé :

my\_tuple = ("jimi", "hendrix")

first\_name, last\_name = my\_tuple

print("First name is: " + first\_name)

print("Last name is: " + last\_name)

First name is jimi

Last name is hendrix

Si c'est le cas, nous pouvons renvoyer un tuple à la fin de la fonction et le décompresser lorsque nous l' *appelons* :

def format\_name(first\_name, last\_name):

return (first\_name.title(), last\_name.title())

first, last = format\_name("RICk", "mORTY")

print(first)

print(last)

Rick

Morty

La fonction reçoit a first\_nameet a last\_name, puis les formate et les renvoie sous forme de tuple.

Lors de l'appel de fonction, le tuple est décompressé en deux variables, firstet last.

**Exercice**

Écrivez une fonction calculation()acceptant deux variables et calculant l'addition et la soustraction. Elle doit également renvoyer l'addition et la soustraction en un seul appel.

For example:

def calculation(a, b):

# Your Code

res = calculation(40, 10)

print(res)

© 2025 **École supérieure de technologie et de commerce de Paris** . Tous