

Fonctions

NSI1

21 octobre 2021

Exemples de fonctions

Nous avons déjà rencontré des fonctions côté utilisateur :

Nous avons déjà rencontré des fonctions côté utilisateur :

- `input`

Nous avons déjà rencontré des fonctions côté utilisateur :

- `input`
 - prend en entrée une chaîne de caractères;

Nous avons déjà rencontré des fonctions **côté utilisateur** :

- **input**
 - prend en entrée une chaîne de caractères;
 - renvoie la chaîne de caractère saisie par l'utilisateur.

Nous avons déjà rencontré des fonctions côté utilisateur :

- `input`
 - prend en entrée une chaîne de caractères;
 - renvoie la chaîne de caractère saisie par l'utilisateur.

On peut noter ceci `input(chaine : str) -> str`

Nous avons déjà rencontré des fonctions côté utilisateur :

- `input`
 - prend en entrée une chaîne de caractères;
 - renvoie la chaîne de caractère saisie par l'utilisateur.

On peut noter ceci `input(chaine : str) -> str`

- `len`

Nous avons déjà rencontré des fonctions côté utilisateur :

- `input`
 - prend en entrée une chaîne de caractères;
 - renvoie la chaîne de caractère saisie par l'utilisateur.

On peut noter ceci `input(chaine : str) -> str`

- `len`
 - prend en entrée une liste;

Nous avons déjà rencontré des fonctions **côté utilisateur** :

- **input**
 - prend en entrée une chaîne de caractères;
 - renvoie la chaîne de caractère saisie par l'utilisateur.

On peut noter ceci **input(chaine : str) -> str**

- **len**
 - prend en entrée une liste;
 - renvoie le nombre d'éléments de cette liste.

Nous avons déjà rencontré des fonctions côté utilisateur :

- `input`

- prend en entrée une chaîne de caractères;
- renvoie la chaîne de caractère saisie par l'utilisateur.

On peut noter ceci `input(chaine : str) -> str`

- `len`

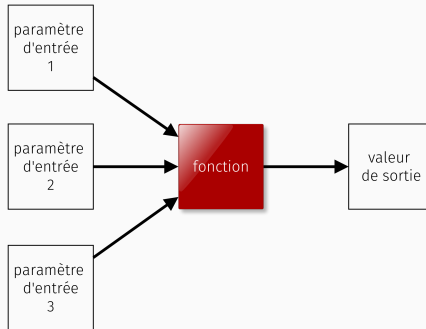
- prend en entrée une liste;
- renvoie le nombre d'éléments de cette liste.

On peut noter cela `len(L : list) -> int`

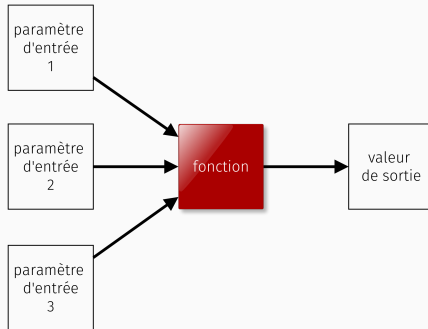
Les deux exemples précédents rentrent dans cette catégorie :



Certaines fonctions sont ainsi :

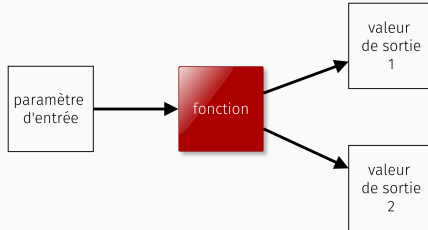


Certaines fonctions sont ainsi :

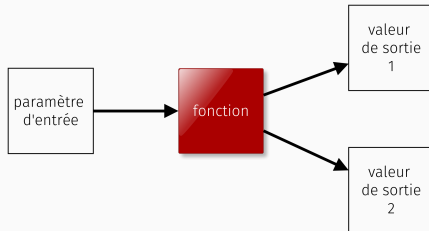


Par exemple `max(20,3,10)` renvoie 20.

D'autres fonctions sont ainsi :

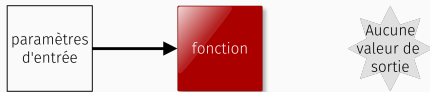


D'autres fonctions sont ainsi :

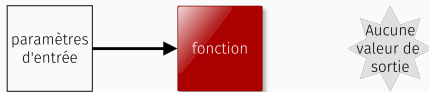


On verra des exemples plus tard.

D'autres encore ainsi :



D'autres encore ainsi :



Par exemple `print("salut")` ne renvoie rien mais affiche `salut` à l'écran.

D'autres suivent ce schéma :



D'autres suivent ce schéma :



Par exemple dans le module `time`, la fonction `time` ne prend aucun paramètre d'entrée mais renvoie l'heure qu'indique l'horloge de l'ordinateur.

D'autres suivent ce schéma :



Par exemple dans le module `time`, la fonction `time` ne prend aucun paramètre d'entrée mais renvoie l'heure qu'indique l'horloge de l'ordinateur.

On peut par exemple l'utiliser pour stocker une heure précise en tapant `maintenant = time()`.

Enfin certaines suivent ce schéma :



Enfin certaines suivent ce schéma :



Par exemple dans le module **p5**, **draw** ne prend aucun paramètre d'entrée, ne renvoie aucune valeur, mais actualise la fenêtre graphique.

Enfin certaines suivent ce schéma :



Par exemple dans le module **p5**, **draw** ne prend aucun paramètre d'entrée, ne renvoie aucune valeur, mais actualise la fenêtre graphique.

Il est possible de créer de nouvelles fonctions.
On parle à ce moment de fonctions **côté concepteur**.

Il faut donc définir rigoureusement ce qu'est une fonction.

Définition de la notion de fonction

Définition

Une *fonction* est un « morceau de code » qui représente un *sous-programme*.

Elle a pour but d'effectuer une tâche *de manière indépendante*.

Un exemple

On veut modéliser la fonction mathématique f définie pour tout nombre réel x par

$$f(x) = x^2 + 3x + 2$$

Un exemple

On veut modéliser la fonction mathématique f définie pour tout nombre réel x par

$$f(x) = x^2 + 3x + 2$$

On écrira alors

```
def f(x : float) -> float:  
    return x ** 2 + 3 * x + 2
```

Un exemple

On veut modéliser la fonction mathématique f définie pour tout nombre réel x par

$$f(x) = x^2 + 3x + 2$$

On écrira alors

```
def f(x : float) -> float:  
    return x ** 2 + 3 * x + 2
```

Pour évaluer ce que vaut $f(10)$ et affecter cette valeur à une variable, on pourra désormais écrire `resultat = f(10)`.

Question

Que fait cette fonction `mystere` ?

```
def mystere(a : float, b : float) -> float:  
    if a <= b:  
        return b  
    else:  
        return a
```

Question

Que fait cette fonction `mystere` ?

```
def mystere(a : float, b : float) -> float:
    if a <= b:
        return b
    else:
        return a
```

La fonction `mystere` :

- prend en entrée deux paramètres de type `float` `a` et `b` ;

Question

Que fait cette fonction `mystere` ?

```
def mystere(a : float, b : float) -> float:  
    if a <= b:  
        return b  
    else:  
        return a
```

La fonction `mystere` :

- prend en entrée deux paramètres de type `float` `a` et `b` ;
- renvoie le plus grand de ces deux nombres.

La réponse que l'on vient de formuler s'appelle *la spécification* de la fonction f .

Définition

Donner la spécification d'une fonction f c'est

Définition

Donner la spécification d'une fonction f c'est

- préciser le(s) type(s) du (des) paramètre(s) d'entrée (s'il y en a);

Définition

Donner la spécification d'une fonction f c'est

- préciser le(s) type(s) du (des) paramètre(s) d'entrée (s'il y en a);
- indiquer sommairement ce que fait la fonction f ;

Définition

Donner la spécification d'une fonction f c'est

- préciser le(s) type(s) du (des) paramètre(s) d'entrée (s'il y en a);
- indiquer sommairement ce que fait la fonction f ;
- préciser le(s) type(s) de la (des) valeur(s) de sortie (s'il y en a).

Anatomie d'une fonction

Exemple

```
def f(L: list) -> int:
    mini = L[0]
    n = len(L)
    for i in range(n):
        if L[i] < mini:
            mini = L[i]
    return mini
```


Exemple

```
def f(L: list) -> int:
    mini = L[0]
    n = len(L)
    for i in range(n):
        if L[i] < mini:
            mini = L[i]
    return mini
```

La fonction `f`

- prend en entrée une liste (sous entendu d'entiers);

Exemple

```
def f(L: list) -> int:
    mini = L[0]
    n = len(L)
    for i in range(n):
        if L[i] < mini:
            mini = L[i]
    return mini
```

La fonction `f`

- prend en entrée une liste (sous entendu d'entiers);
- renvoie le plus petit entier de cette liste.

Anatomie de la fonction : paramètre formel

```
def f(L: list) -> int:
```

```
    mini = L[0]
```

```
    n = len(L)
```

```
    for i in range(n):
```

```
        if L[i] < mini:
```

```
            mini = L[i]
```

```
    return mini
```

Paramètre
formel



Anatomie de la fonction : paramètre formel

```
def f(L: list) -> int:
```

```
    mini = L[0]
```

```
    n = len(L)
```

```
    for i in range(n):
```

```
        if L[i] < mini:
```

```
            mini = L[i]
```

```
    return mini
```

Paramètre
formel



Le paramètre d'entrée est *formel* : le nom de cette variable n'existe qu'à l'intérieur de la fonction.

Anatomie de la fonction : paramètre formel

```
def f(L: list) -> int:
```

```
    mini = L[0]
```

```
    n = len(L)
```

```
    for i in range(n):
```

```
        if L[i] < mini:
```

```
            mini = L[i]
```

```
    return mini
```

Paramètre
formel




Le paramètre d'entrée est *formel* : le nom de cette variable n'existe qu'à l'intérieur de la fonction.

Si ce nom de variable existe déjà à l'extérieur de la fonction, ce n'est pas la même variable.

Anatomie de la fonction : paramètre formel

```
def f(L: list) -> int:  
    mini = L[0]  
    n = len(L)  
    for i in range(n):  
        if L[i] < mini:  
            mini = L[i]  
    return mini
```


Type du
paramètre
d'entrée



Anatomie de la fonction : paramètre formel

```
def f(L: list) -> int:  
    mini = L[0]  
    n = len(L)  
    for i in range(n):  
        if L[i] < mini:  
            mini = L[i]  
    return mini
```

Type du
paramètre
d'entrée



Le type du paramètre d'entrée peut être spécifié. Ce n'est pas obligatoire mais très fortement recommandé pour « garder les idées claires ».

Anatomie de la fonction : variables locales

```
def f(L: list) -> int:
```

```
    mini ← L[0]
```

```
    n ← len(L)
```

```
    for i in range(n):
```

```
        if L[i] < mini:
```

```
            mini ← L[i]
```

```
    return mini
```

Variables
locales

A diagram illustrating local variables. On the right, the text "Variables locales" is written. On the left, within a light gray box, is a Python function definition. Arrows point from the following variables in the code to the text: 'mini' in the first assignment, 'n' in the second assignment, 'i' in the for loop, 'mini' in the if condition, 'mini' in the inner assignment, and 'mini' in the return statement.

Anatomie de la fonction : variables locales

```
def f(L: list) -> int:
```

```
    mini ← L[0]
```

```
    n ← len(L)
```

```
    for i in range(n):
```

```
        if L[i] < mini:
```

```
            mini ← L[i]
```

```
    return mini
```

Variables
locales



Toutes les variables **créées** dans une fonction n'existent **que dans cette fonction**.

Anatomie de la fonction : variables locales

```
def f(L: list) -> int:
```

```
    mini ← L[0]
```

```
    n ← len(L)
```

```
    for i in range(n):
```

```
        if L[i] < mini:
```

```
            mini ← L[i]
```

```
    return mini
```

Variables
locales



Toutes les variables **créées** dans une fonction n'existent **que dans cette fonction**. Elles ne sont pas accessibles depuis l'extérieur de la fonction. On dit que ce sont des **variables locales**.

Anatomie de la fonction : valeur de sortie

```
def f(L: list) -> int:  
    mini = L[0]  
    n = len(L)  
    for i in range(n):  
        if L[i] < mini:  
            mini = L[i]  
    return mini
```

Type de la
valeur de sortie

Anatomie de la fonction : valeur de sortie

```
def f(L: list) -> int:  
    mini = L[0]  
    n = len(L)  
    for i in range(n):  
        if L[i] < mini:  
            mini = L[i]  
    return mini
```

Type de la
valeur de sortie

Le type de la valeur de sortie peut être précisé, c'est également recommandé.

En pratique

```
1 def f(x : float) -> float:
2     return x ** 2 + 3 * x + 2
3
4 print(f(1)) # Affiche 6
```

```
1 def f(x : float) -> float:
2     return x ** 2 + 3 * x + 2
3
4 print(f(1)) # Affiche 6
```

Le programme commence à la ligne 4!

Les 2 premières lignes servent à définir la fonction **f**, elles ne sont exécutées que lorsqu'on évalue **f(1)**.

Exemple 1

```
def f(x : float) -> float:  
    return x ** 2 + 3 * x + 2  
  
print(x) # Provoque une erreur
```


Exemple 1

```
def f(x : float) -> float:
    return x ** 2 + 3 * x + 2

print(x) # Provoque une erreur
```

L'erreur vient du fait que la variable x **n'est pas définie**. Le « x qu'on voit dans la fonction f » est un paramètre formel et n'existe que dans f .

Exemple 2

```
def f(x : float) -> float:  
    a = 2  
    return x + a  
  
print(a) # Provoque une erreur
```

Exemple 2

```
def f(x : float) -> float:  
    a = 2  
    return x + a  
  
print(a) # Provoque une erreur
```

L'erreur vient du fait que la variable **a** est locale : elle n'est définie que durant l'exécution de **f**.

Exemple 3

```
def f(x : float) -> float:  
    a = 2  
    return x + a  
  
print(f(4)) # Affiche 6  
print(a) # Provoque une erreur
```

Exemple 3

```
def f(x : float) -> float:  
    a = 2  
    return x + a  
  
print(f(4)) # Affiche 6  
print(a) # Provoque une erreur
```

C'est encore la même erreur : une fois $f(4)$ évaluée, a n'existe plus.

Exemple 4

```
1 def f(x : float) -> float:
2     a = 2
3     return x + a
4
5 a = 3
6 print(f(4)) # Affiche 6
7 print(a) # Affiche 3 et pas 2
```

Exemple 4

```
1 def f(x : float) -> float:
2     a = 2
3     return x + a
4
5 a = 3
6 print(f(4)) # Affiche 6
7 print(a) # Affiche 3 et pas 2
```

La variable `a` définie dans la fonction `f` n'est pas la même que celle qui est définie à la ligne 5.

Celle définie à la ligne 2 est **locale**.

La variable `a` de la ligne 5 est appelée **globale**.

Exemple 4

```
1 def f(x : float) -> float:
2     return x + a
3
4 a = 3
5 print(f(4)) # Affiche 7
```


Exemple 4

```
1 def f(x : float) -> float:
2     return x + a
3
4 a = 3
5 print(f(4)) # Affiche 7
```

À retenir

Une fonction a le droit d'**accéder en lecture** à une variable globale, mais n'a pas *a priori* le droit d'en modifier la valeur.

```
1 def f(x : float) -> float:
2     global a
3     a = a + 1
4     return x + a
5
6 a = 3
7 print(f(4)) # Affiche 8
8 print(a) # Affiche 4
```

À éviter

```
1 def f(x : float) -> float:
2     global a
3     a = a + 1
4     return x + a
5
6 a = 3
7 print(f(4)) # Affiche 8
8 print(a) # Affiche 4
```

À la ligne 2, on signale à Python que `f` a le droit de modifier la variable globale `a`.

C'est fortement déconseillé : sauf si on ne peut pas faire autrement, une fonction ne doit pas modifier les variables globales.