

Fonctions : notions avancées

Fonctions

Les fonctions sont *first-class objects*, objets comme les autres (comme les entiers, listes, dictionnaires, ...)

On fait référence à l'objet-fonction par son **nom sans parenthèses**.

```
>>> def f(x) :  
.....     return x + 1  
>>> f(4) # appel de la fonction  
5  
>>> f     # ceci n'est pas un appel !  
<function f at 0x7fd3f71ccf28>
```

Fonctions

Si on regarde une fonctions f comme étant une *machine* :

$f()$

appeler la fonction = *utiliser* la machine

f

parler de la fonction/machine sans l'exécuter

Fonctions

On peut faire tout ce qu'on fait avec les autres objets :

- stocker dans variables
- passer en argument aux autres fonctions

```
def incrementer(x) :
```

```
    return x + 1
```

```
def dupliquer(x) :
```

```
    return x * 2
```

```
f = dupliquer      # sans parenthèses : pas d'exécution
```

```
print(f(3))        # affiche : 6
```

```
f = incrementer    # sans parenthèses : pas d'exécution
```

```
print(f(3))        # affiche : 4
```

Fonctions

```
def incrementer(x):  
    return x + 1  
def dupliquer(x):  
    return x * 2
```

```
# on collecte le résultat de l'exécution  
liste1 = [incrementer(3), dupliquer(3)]
```

```
# on collecte les fonctions elles-mêmes  
liste2 = [incrementer, dupliquer]
```

Fonctions

```
from turtle import *
```

```
def poly(n, size):  
    for _ in range(n):  
        forward(size)  
        left(360 / n)
```

```
def etoile(n, size):  
    for _ in range(n):  
        left(20)  
        forward(size)  
        right(2 * 20)  
        forward(size)  
        left(20 + 360 / n)
```

Fonctions

```
from turtle import *
```

```
def poly(n, size):  
    for _ in range(n):  
        forward(size)  
        left(360 / n)
```

```
def etoile(n, size):  
    for _ in range(n):  
        left(20)  
        forward(size)  
        right(2 * 20)  
        forward(size)  
        left(20 + 360 / n)
```

Même signature !



Fonctions

```
from random import randint
```

```
def random_figure(shape) :  
    x = randint(-200, 200)  
    y = randint(-200, 200)  
    up() ; goto(x, y) ; down()  
    n = randint(3, 8)  
    size = randint(10, 50)  
    shape(n, size)
```

```
random_figure(etoile)
```

```
random_figure(poly)
```


Fonctions

```
# une liste de fonctions
liste = [etoile, poly]

# un élément est choisi au hasard
s = random.choice(liste)

random_figure(s)
```

en résumé...

$f()$

Ceci n'est pas une fonction

Arguments par défaut

Fonctions

On peut donner des valeur par défaut aux arguments avec la syntaxe

```
def fonction(variable=valeur) :  
    [...]
```

```
>>> def f(x=2) :  
...     print(x**2)  
...
```

```
>>> f(5)
```

```
25
```

```
>>> f()    # pas d'argument : x=2 utilisé par défaut
```

```
4
```

Fonctions

Les arguments traditionnels peuvent coexister avec les arguments prédéfinis.

Attention ! Les arguments prédéfinis doivent *suivre* les autres :

```
def f(x, y=0):  
    print(x*y)
```



```
def f(x=0, y):  
    print(x*y)
```



Fonctions

Pour éviter des ambiguïtés :

```
>>> def f(x=0, y, z=1) :  
...     print(x * y + z)  
>>> f(3, 5)
```

$x=3, y=5$?
 $y=3, z=5$?

Fonctions

On peut passer les arguments *par nom* ([keyword arguments](#)).

```
>>> def f(x, y, z) :  
...     print(x + y + z)  
>>> f(z=1, x=2, y=3)  
6
```

Si on passe à la fois des arguments normales et des *keyword arguments*, ces derniers doivent suivre les autres.

```
>>> f(2, z=1, y=3)  
6
```

Exemple : la fonction `print()` et son argument `end=`

Exemple : la fonction `max()`

Deux usages : `max(iterable)` ou `max(a, b, c, ...)`

```
>>> max([1, 2, 3])
```

```
3
```

```
>>> liste = ['aaa', 'bb', 'z']
```

```
>>> max(liste)
```

```
'z'
```

```
>>> max(3.3, 2.5, 4.0, 1.2)
```

```
4.0
```

Exemple : la fonction `max()`

`max()` accepte un *keyword argument* optionnel `key`.

`key` doit être une fonction, utilisée pour comparer les objets.

```
>>> max('aaa', 'bb', 'z')
```

```
'z'
```

```
>>> max('aaa', 'bb', 'z', key=len)
```

```
'aaa'
```

Les chaînes sont comparées en utilisant la valeur de `len()`:

```
len('aaa') = 3
```

```
len('bb') = 2
```

```
len('z') = 1
```

Exemple : la fonction `max()`

On n'est pas limité aux fonctions prédéfinies.

```
>>> def neg(x) :  
...     return -x  
...  
>>> liste = [1, 2, 3]  
>>> max(liste, key=neg)  
1
```

Exemple : la fonction `sorted()`

Créer une nouvelle liste triée sans changer l'originale.

```
>>> liste = [3, 1, 2]
```

```
>>> sorted(liste)
```

```
[1, 2, 3]
```

Exemple : la fonction `sorted()`

Comme pour la fonction `max()` :

argument `key=`

```
>>> liste = ['bb', 'z', 'aaa']
```

```
>>> sorted(liste)
```

```
['aaa', 'bb', 'z']
```

```
>>> sorted(liste, key=len)
```

```
['z', 'bb', 'aaa']
```

Fonctions

Comment trier un ensemble des tuple selon le dernier élément ?

```
>>> def dernier(t):  
.....     return t[-1]  
>>>  
>>> liste = [(1, 5), (3, 5, 1),  
(3, 3), (4, 2)]  
>>> sorted(liste, key=dernier)  
[(3, 5, 1), (4, 2), (3, 3), (1, 5)]
```

Fonctions anonymes (*lambda*)

Fonctions anonymes

Pour des fonctions très courtes, on peut utiliser des **fonctions anonymes**, connues aussi comme **expressions lambda**.

Revenons à **def**...

Le mot-clé **def** fait deux choses :

1. créer la fonction (donc : créer un objet)
2. donner un nom à cet objet

```
def dernier(t) :  
    return t[-1]
```

1. créer la fonction $t \mapsto t[-1]$
2. appeler cette fonction « dernier »

Fonctions anonymes

Pour 1. (créer l'objet-fonction) il existe aussi le mot-clé **lambda** :

lambda $t : t[-1]$

Si on veut, on peut attribuer un nom par affectation :

dernier = **lambda** $t : t[-1]$

Fonctions anonymes

L'exemple du triage de listes avec une expression lambda :

```
>>> liste = [(1, 5), (3, 5, 1), (3, 3),  
             (4, 2)]  
>>> sorted(liste, key=lambda t: t[-1])  
[(3, 5, 1), (4, 2), (3, 3), (1, 5)]
```

Fonctions anonymes

En générale :

```
lambda var1, var2, ..., varN: une_expression
```

```
>>> f = lambda x, y: x + y
```

```
>>> f(3, 5)
```

```
8
```

Fonctions anonymes

Dans le corps de la fonction anonyme :

- pas de **return**, l'expression est automatiquement retournée
- pas d'instructions, seulement **une** expression
(par exemple : `for`, `if`, `while` ne sont admis !)


Nombre variable d'arguments

Fonctions

Pour avoir un nombre variable d'arguments on utilise
l'emballage de tuple : un seul argument préfixé par '*'

```
def f(*args):  
    for x in args:  
        print(x)
```

Pas de '*' ici !



f(2)

f(2, 4)

f(2, 4, 6)

Dans l'exemple, `args` devient un tuple qui contient tous les arguments passés à `f()`.

Fonctions

```
def moyenne (*nombres) :  
    m = sum(nombres) / len(nombres)  
    return m
```

```
f(2)                # résultat : 2
```

```
f(2, 4)             # résultat : 3
```

```
f(2, 4, 6)          # résultat : 4
```

Fonctions

Les arguments emballés en tuple peuvent coexister avec les autres (mais... ils doivent suivre !)

```
def g(a, b, *args) :  
    print(args)
```

```
g(1, 2)
```

```
g(1, 2, 3, 4, 5)
```


Fonctions

Le même opérateur `*` (utilisé dans une appelle de fonction) peut *déballer* un tuple ou une liste.

```
>>> def f(a, b, c):  
...     print('a={}, b={}, c={}'.format(a, b, c))  
...  
>>> t = (1, 2, 3)  
>>> f(*t)  
a=1, b=2, c=3  
>>> l = [2, 3, 4]  
>>> f(*l)  
a=2, b=3, c=4
```

Fonctions

Emballages de **keyword arguments** dans un dictionnaire.

```
>>> def f (**kwargs) :  
...     print (kwargs)  
...  
>>> f (x=5,  couleur='bleu')  
{ 'x' : 5, 'couleur' : 'bleu' }
```

Fonctions

A l'envers :

```
>>> def f(a, b, c):  
...     print('a={}, b={}, c={}'.format(a, b, c))  
...  
>>> d = {'a' : 1, 'b' : 2, 'c' : 3}  
>>> f(**d)  
a=1, b=2, c=3
```

Portée des variables

Portée des variables

Les variables créées dans une fonction, sont **locales**.

La variable `x` est **locale** pour la fonction `f`.

```
>>> def f():  
...     x = 5  
...     print(x)  
...
```

```
>>> f()
```

```
5
```

```
>>> print(x)
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
NameError: name 'x' is not defined
```

Portée des variables

Les variables **globales** sont visibles dans les fonctions.

```
>>> def f():  
...     print(x)  
...  
>>> x = 5  
>>> f()  
5
```

Pourvu que la fonction soit exécutée après la définition de la variable **globale**.

Portée des variables

Ombrage des variables :

Les variables locales peuvent cacher des variables globales qui ont le même nom.

```
>>> def f():  
...     x = 2  
...     print(x)  
...  
>>> x = 5  
>>> f()  
2  
>>> print(x)  
5
```

L'affectation **`x = 2`** dans la fonction `f` crée une **nouvelle variable locale**.

La variable globale du même nom `x` reste cachée (pour `f`).

Portée des variables

```
>>> def f() :  
...     global x  
...     x = 2  
...     print(x)  
...  
>>> x = 5  
>>> f()  
2  
>>> print(x)  
2
```

L'instruction `global x` permet d'utiliser et changer des variable globales.
À utiliser avec modération !

Portée des variables

```
>>> def f():  
...     print(x)  
...     x = 2  
...     print(x)  
...  
>>> x = 5  
>>> f()
```

Portée des variables

```
>>> def f():  
...     print(x)  
...     x = 2  
...     print(x)  
...
```

```
>>> x = 5
```

```
>>> f()
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
  File "<stdin>", line 2, in f
```

```
UnboundLocalError: local variable 'x' referenced before  
assignment
```

L'affectation **x = 2** rend **x locale dans toute la fonction**. La première instruction `print()` essaie d'évaluer une variable locale qui n'a pas (encore) de valeur.

Portée statique

La portée des variable est *statique* (synonymes : *lexicale*, *textuelle*).

Lors de la définition d'une fonction, Python détermine la portée récursivement.

```
def f() :  
    print(a)
```

```
def g() :  
    a = 5  
    f()
```

```
g()
```

Portée statique

```
def f() :  
    print(a) # 'a' n'est pas locale  
             # --> 'a' cherchée dans le scope  
             #       supérieur (le scope global)  
  
def g() :  
    a = 5  
    f()  
  
g()      # NameError ! 'a' n'est pas définie dans le  
         # scope global
```

Portée statique

Par contre :

```
def g() :  
    a = 5  
    def f() :  
        print a    # 'a' n'est pas locale  
                   # --> 'a' cherchée dans le  
                   #       scope supérieur (celui  
                   #       de 'g')  
    f()  
  
g()    # Affiche '5'
```

Fonctions imbriquées

Définition de fonctions imbriquées

```
def f(n) :  
    def square(x) :  
        return x**2  
  
    for i in range(n) :  
        print (square(i))
```

Fonctions imbriquées

On peut renvoyer la fonction ainsi définie !

```
def make_incrementer() :  
    def incrementer(x) :  
        return x + 1  
    return incrementer
```

```
f = make_incrementer()  
print(f(4))                # Affiche : 5
```

Fonctions imbriquées

Même structure :

```
def make_incrementer() :  
    def incrementer(x) :  
        return x + 1  
    return incrementer
```

```
def make_list() :  
    liste = [1, 2, 3]  
    return liste
```


Closures

Les principal cas d'utilisation pour les fonctions imbriquées est créer des *closures* (ou *fermeture*, *clôture*)

Closure = fonction avec un « contexte » de variables affectées

```
def make_adder(n) :  
    def adder(x) :  
        return x + n  
    return adder
```

```
f = make_adder(3)  
print(f(5))          # Affiche : 8
```

La fonction créé par `make_adder(3)` est un fonction d'une variable (`x`),
avec un « contexte » (`n = 3`)

Closures

```
def make_greeter(nom) :  
    s = 'Hello, ' + nom  
    def greeter():  
        print(s)  
    return greeter
```

```
g1 = make_greeter('world')  
g2 = make_greeter('moon')
```

```
g1()    # affiche 'Hello, world'  
g2()    # affiche 'Hello, moon'
```