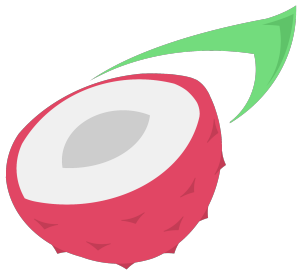

Litchi Pi

Python

Utilisation avancée des fonctions

6 Novembre 2025

Rappel sur les fonction

Fonction = Code réutilisable à chaque appel, pouvant prendre des *arguments*

```
def my_function(my_argument):
  return my_argument * 2
```

En python les arguments ne sont pas *typés* par défaut, donc **attention** aux erreurs

Il est aussi possible de passer des valeurs par défaut

```
def my_function(my_argument, other_arg=3):
  return my_argument * other_arg

my_function(6) # Retourne 18
my_function(3, other_arg=4) # Retourne 12
```

En Python, une fonction est une *instance de classe*, c'est un **objet**

```python
class MyFunction:
  def __init__(self, x):
    self.x = x

  def __call__(self):
    return self.x * 2

  def __repr__(self):
    return "Ma petite fonction <3"

f = MyFunction(3)
print(f()) # Affiche 6
print(f)   # Affiche "Ma petite fonction <3"
```

Fonctions récursives

Récursive = Fonction qui s'appelle elle-même

Créé un problème s'il n'y a pas de **condition de sortie**

En Python, créé un `RecursionError` au bout de 1000 répétition.

```python
def recurs_f(x, nb):
  # Condition de sortie
  if nb <= 0:
    return x

  return recurs_f(x+nb, nb-1)
```

*args et **kwargs

Nombre **variable** d'arguments ?

`*args` récupère les *arguments anonymes* sous forme de *tuple*

`**kwargs` récupère les *arguments nommés* sous forme de *dict*

```
def my_function(*args, **kwargs):
  print(args)
  print(kwargs)

my_function(3, 5, 8, test="toto", iwant="kebab")
# Affiche ( 3, 5, 8 )
#    puis { 'test': 'toto', 'iwant': 'kebab' }
```

Peut être utilisé aussi en inverse

```
my_args = (3, 2, 1)
my_function(*my_args)
# Affiche ( 3, 2, 1 )

my_kwargs = { "name": "Diego", "description": "Grandes dents" }
my_function(**my_kwargs)
# Affiche { 'name': 'Diego', 'description': 'Grandes dents' }

my_function(*my_args, **my_kwargs)
```

Utilisation mixte

```
def my_function(x, *inp, other=3, **cfg):
  return x * other

my_function(3)
my_function(3, 4, 5)
my_function() # Erreur
my_function(4, other=2, toto=1)

my_config = { 'other': 4, 'verbose': True }
my_function(4, 8, 3, **my_config)
```

Fonction en tant que variable

On peut assigner une fonction à une variable

```
def default_handler(ip_address, **details):
  pass

class WebServer:
  def __init__(self):
    self.handler = default_handler

  def handle_connection(self, ip_address, connection_details):
    self.handler(ip_address, **connection_details)
```

```
def my_handler(ip_address, **details):
  print("My own handler")
  pass

srv = WebServer()
srv.handler = my_handler
```

Fonction en tant que variable

```
def new_cmd(*args, **kwargs):
  pass

def edit_cmd(*args, **kwargs):
  pass

def view_cmd(*args, **kwargs):
  pass

HANDLERS = {
  "new":  new_cmd,
  "edit": edit_cmd,
  "view": view_cmd,
}

if cmd in HANDLERS:
  HANDLERS[cmd]()
else:
  print("Command not found")
```

Décorateurs

Comment **modifier l'appel** à une fonction "on the fly" ?

- Pré-processing des inputs d'une fonction
- Mesurer le temps qu'a mis la fonction
- Afficher informations de debug

```
def my_decorator(f):
  # Créé la fonction transformée
  def decorated(*args, **kwargs):
    print("Hello, world!")

    # Appelle la fonction de base
    return f(*args, **kwargs)

  # Retourne la fonction transformée
  return decorated
```

```
@my_decorator
def my_function(x):
  return x ** 2
```

```python
def my_decorator(f):
  # Créé la fonction transformée
  def decorated(*args, **kwargs):
    t = time.time()
    print(f"Function start, with *args {args}, and kwargs {kwargs}")

    # Appelle la fonction de base
    ret = f(*args, **kwargs)

    dt = time.time() - t
    print(f"Function end, took {dt}secs")

    return ret

  # Retourne la fonction transformée
  return decorated
```

Vous pouvez aussi **décorer des classes**, voir *exemple sur StackOverflow*

TP

Décorer l’algorithme génétique