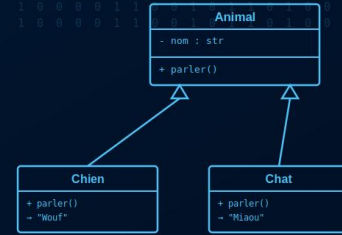


III. Fonctionnalités avancées



héritage (POO)



```

class Animal:
    def __init__(self, nom):
        self.nom = nom
    def parler(self):
        raise NotImplementedError

class Chien(Animal):
    def parler(self):
        return "Wouf !"
    
```

M. Martin

1

1

Problème

Comment **structurer un projet complet** avec plusieurs modules ?

Solution

- Un **paquetage** est un dossier contenant plusieurs modules
- Il est identifié par un fichier « `__init__.py` » qui peut être vide ou contenir du code d'initialisation

Arborescence typique

```

mon_paquetage/
|-- __init__.py
|-- module1.py
|-- module2.py
    
```

Utilisation

```

from mon_paquetage import module1
    
```

2

2

Lecture ligne par ligne

```
with open("data/capteurs.txt", "r", encoding="utf-8") as f:
    for ligne in f:
        print(ligne.strip())
```

Lecture complète

```
with open("data/capteurs.txt", "r") as f:
    contenu = f.read()
```

Écriture

```
with open("results/out.txt", "w", encoding="utf-8") as f:
    f.write("Bonjour\n")
```

- **Modes d'ouverture** : "r" lecture; "w" écriture (efface le fichier); "a" ajout; "b" binaire (utile pour images ou données brutes)

3

3

Un **générateur** est une fonction qui produit les valeurs au fur et à mesure, sans stocker toute la séquence en mémoire.

- Il utilise le mot-clé **yield** au lieu de **return**
- Chaque appel de l'itérateur reprend l'exécution là où elle s'était arrêtée

Exemple

```
def compte(n):
    for i in range(n):
        yield i

for x in compte(5):
    print(x) # Affiche 0, 1, 2, 3, 4
```

Les **expressions génératrices** fournissent une syntaxe compacte pour créer des générateurs (assez proche des compréhensions).

4

4

Problème

Comment lire un énorme fichier (plusieurs Go) ou encore générer des millions de points ?

Solution

- Tout stocker dans une liste est impossible
- Il faut **utiliser un générateur** : seules les valeurs nécessaires sont produites « à la volée »

Comparaison mémoire

```
import sys
liste = [x for x in range(10**6)]
gen = (x for x in range(10**6))
print(sys.getsizeof(liste)) # taille en mémoire de la liste
print(sys.getsizeof(gen)) # taille fixe du générateur
```

5

5

Un **décorateur** modifie ou enrichit le comportement d'une fonction sans changer son code.

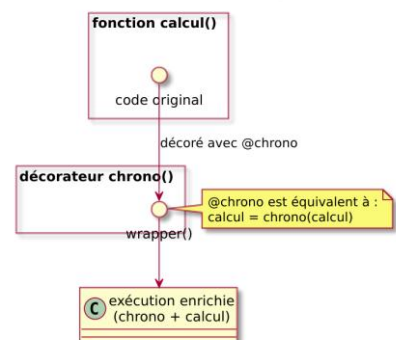
Usage fréquent : journalisation, contrôle des entrées, tests, etc...

Exemple : mesurer le temps d'exécution

```
import time
def chrono (fonction):
    def wrapper(*args, **kwargs):
        debut = time.time()
        resultat = fonction(*args, **kwargs)
        fin = time.time()
        print("Durée : ", fin - debut, "s")
        return resultat
    return wrapper

@chrono
def calcul():
    return sum(x**2 for x in range(10**6))
calcul()
```

Mécanisme d'un décorateur (@chrono)



6

6

Gestion des contextes (mot-clé with)

Les objets contextuels permettent une gestion propre des ressources (exemples : fichiers, connexions réseau, verrous).

- Le mot-clé **with** garantit une **libération correcte des ressources**

```
with open("data/capteurs.txt", "r") as f:
    contenu = f.read()
```

- est équivalent à :

```
f = open("data/capteurs.txt", "r")
try:
    contenu = f.read()
finally:
    f.close()
```

7

7

Copie de structures en Python

Problème

En Python, l'opérateur = ne crée pas une copie mais une nouvelle référence :

```
a = [1, 2, 3]
b = a
b[0] = 99
print(a) # [99, 2, 3] -> a est aussi modifié !
```

Solutions

1. Copie superficielle (shallow copy)

Une nouvelle structure est créée, mais les éléments restent partagés si ce sont eux-mêmes des conteneurs (liste, tuple, ensemble, dictionnaire)

```
import copy
a = [[1, 2], [3, 4]]
b = copy.copy(a)
b[0][0] = 99
print(a) # [[99, 2], [3, 4]] -> éléments internes partagés
```

8

8

Copie de structures en Python

Problème

En Python, l'opérateur = ne crée pas une copie mais une nouvelle référence :

```
a = [1, 2, 3]
b = a
b[0] = 99
print(a) # [99, 2, 3] -> a est aussi modifié !
```

Solutions

2. Copie profonde (deep copy)

Toute la structure est dupliquée récursivement :

```
import copy
a = [[99, 2], [3, 4]]
b = copy.deepcopy(a)
b[0][0] = -1
print(a) # [[99, 2], [3, 4]]
print(b) # [[-1, 2], [3, 4]]
```

9

9

Modules utiles de la bibliothèque standard

- **os** : interaction avec le système de fichiers
- **pathlib** : gestion moderne des chemins
- **sys** : paramètres et gestion de l'interpréteur
- **time, datetime** : gestion du temps
- **random** : génération pseudo-aléatoire
- **json** : manipulation de fichiers JSON
- **re** : expressions régulières
- **collections** : structures avancées (ex. Counter, defaultdict)

```
from pathlib import Path
import datetime, random, collections

print(Path.cwd()) # Dossier courant
print(datetime.date.today()) # Date du jour
print(random.randint(1, 6)) # Entier aléatoire entre 1 et 6
compte = collections.Counter("abracadabra")
print(compte) # Compte l'occurrence des lettres
```

10

10

Python permet de passer des fonctions comme arguments :

```
def square(x):
    return x ** 2

def cube(x):
    return x ** 3

def transform_numbers(number_list, transform):
    number0 = transform(number_list[0])
    number1 = transform(number_list[1])
    return [number0, number1]

numbers = [2, 5]
squares = transform_numbers(numbers, square) # [4, 25]
cubes = transform_numbers(numbers, cube) # [8, 125]
```

11

11

C'est une fonction **anonyme** constituée **d'une unique instruction** :

```
lambda paramètre(s): opérations
```

Exemples

- $x \rightarrow x^2$

```
lambda x: x ** 2
```

- $x, y \rightarrow (x + y)^2$

```
lambda x, y: (x + y) ** 2
```

Utilisation dans l'exemple vu plus haut :

```
def transform_numbers(number_list, transform):
    number0 = transform(number_list[0])
    number1 = transform(number_list[1])
    return [number0, number1]

numbers = [2, 5]
squares = transform_numbers(numbers, lambda x: x ** 2)
cubes = transform_numbers(numbers, lambda x: x ** 3)
```

12

12

La fonction `map()` applique une fonction à un itérable.

Syntaxe

```
map(function, iterable)
```

Exemple

```
numbers = [1, 2, 3, 4]
print(list(map(lambda x: x ** 3, numbers))) # cubes = [1, 8, 27, 64]
```

La fonction `filter()` filtre une liste en vérifiant la condition d'une fonction.

Syntaxe

```
filter(function, iterable)
```

Exemple

```
numbers = [2, 3, 4, 5, 6, 7]
print(list(filter(lambda x: x % 2 == 0, numbers))) # even_numbers = [2, 4, 6]
```

13

13

Python intègre plusieurs outils du paradigme « fonctionnel » :

- `map()` et `filter()` (voir plus haut)
- Fonctions anonymes `lambda` (voir plus haut)
- Modules `itertools`, `functools` et sa fonction `reduce()`

Exemple

```
from functools import reduce

numbers = [1, 2, 3, 4]
print(list(map(lambda x: x ** 2, numbers))) # [1, 4, 9, 16]
print(list(filter(lambda x: x % 2 == 0, numbers))) # [2, 4]
print(reduce(lambda x, y: x + y, numbers)) # 10
```

14

14

Exemple avec itertools

```
import itertools

for x in itertools.permutations([1, 2, 3]):
    print(x)
for x in itertools.islice(itertools.cycle("AB"), 6):
    print(x)  # A B A B A B
```