EPFL

ICC

Programmation

GC/MX, **Cours 8**

11 novembre 2022

Jean-Philippe Pellet

Opening music: Bob Dylan, album *Blonde on Blonde* (1966)

Previously, on Programmation…

- Types de base en Python: int, float, str, bool
- Méthodes, fonctions et slicing pour calculer des valeurs dérivées
- Conditions pour exécuter du code selon la valeur d'une expression booléenne: `if <condition>: ... else: ...` et ses variantes
- Boucles pour exécuter du code plusieurs fois:
  - Boucle `while <condition>: ...`
  - Boucle `for i in range(...): ...`
- Déclaration de fonctions avec type de retour et paramètres:
  - `def calculate_area(r: float) -> float: return ...`
- Utilisation de listes
- Utilisation de sets
- Utilisation de dictionnaires
  - Ce sont tous des objects modifiables

Seconde partie du semestre

- Cours
  - Comme d'habitude
  - Exception: absence le vendredi 9 décembre
    - ➡ Pas de cours synchrone en CM1, mais vidéo à regarder
    - ➡ Exercices comme d'habitudes avec les assistants
- Exercices
  - Semaine prochaine (18 novembre), début du miniprojet
  - Travail par groupes de deux
  - Vous avez 3 semaines pour travailler dessus, rendu au plus tard le 9 décembre à 23h59
  - Séances d'exercices en parallèle (mais plus courtes pour laisser du temps au miniprojet)

Miniprojet

Selon une idée et projet original de Jamila Sam et Barbara Jobstmann

Original

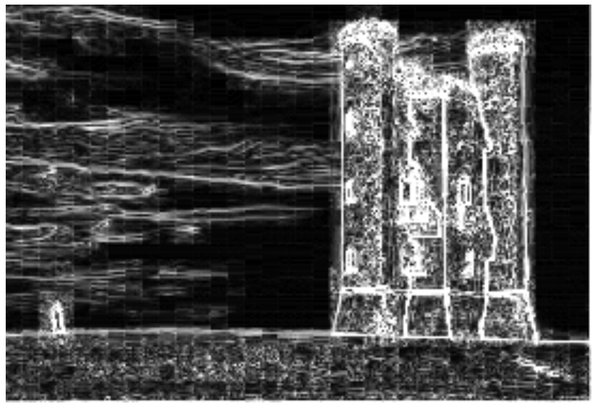
Étirée

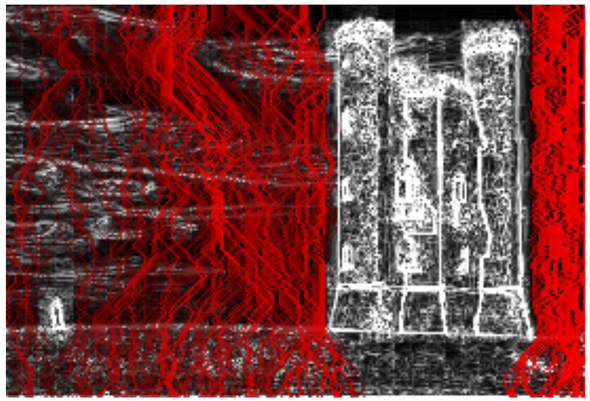
Rognée

Redimensionnée avec la méthode du seam carving

Miniprojet

Original





Résultat

Détection des pixels qui sont porteurs de plus ou moins d'information par rapport à leurs voisins

Détection de tracés de haut en bas qui passent par des pixels de basse information

Suppression de ces tracés dans l'image originale

Cours de cette semaine

Dictionnaires

Classes simples

Motivation

- Classes:
  «Je veux modéliser des types plus complexes (par exemple un objet Cylinder) et rassembler les données et opérations y relatives en un seul endroit»

Classe = type plus avancé

Une classe modélise un objet de la vie réelle (ou un concept abstrait)

Une classe est un modèle pour un objet, en définissant ses données (les variables qui l'accompagnent) et ses opérations et calculs (ses méthodes)

Démo

Exemple suivi:
calcul du volume et de l'aire d'un cylindre

Étape 0: sans classe, avec fonctions

```python
import math

def calc_cylinder_volume(radius: float, height: float) -> float:
    return math.pi * radius * radius * height

def calc_cylinder_surface_area(height: float, radius: float) -> float:
    a1 = 2 * math.pi * radius * height
    a2 = 2 * math.pi * radius * radius
    return a1 + a2

r = 1.2
h = 3.5
v = calc_cylinder_volume(r, h)
a = calc_cylinder_surface_area(r, h)
print(f"v = {v}, a = {a}")
```

Les fonctions, avec arguments demandés et type de retour comme on les connaît déjà

Les valeurs qui définissent le cylindre

Les appels de fonctions qui font les calculs, en leur passant les arguments

Tout semble OK, mais il y a une erreur. Où est-elle?

Étape 1: avec classe + fonctions

```
def calc_cylinder_volume(cyl: Cylinder) -> float:
    return math.pi * cyl.radius * cyl.radius * cyl.height

def calc_cylinder_surface_area(cyl: Cylinder) -> float:
    a1 = 2 * math.pi * cyl.radius * cyl.height
    a2 = 2 * math.pi * cyl.radius * cyl.radius
    return a1 + a2

cyl = Cylinder(1.2, 3.5)
v = calc_cylinder_volume(cyl)
a = calc_cylinder_surface_area(cyl)
print(f"v = {v}, a = {a}")
```

Les fonctions ne demandent plus qu'un argument de type *Cylinder*. Plus de risque de confusion entre r et h!

Les fonctions utilisent la notation *objet.champ* pour récupérer les valeurs stockées par un *Cylinder*. Un *champ* est une «sous-variable»

Une variable d'un nouveau type, *Cylinder*, défini par nous! Il stocke le rayon et la hauteur ensemble

«Je suis une nouvelle classe, *Cylinder*»

Chaque cylindre stocke deux floats comme «sous-variables». Le premier s'appelle *radius*, le deuxième *height*

```
from dataclasses import dataclass

@dataclass
class Cylinder:
    radius: float
    height: float
```

Construction avec noms des paramètres

```python
from dataclasses import dataclass

@dataclass
class Cylinder:
    radius: float
    height: float


cyl = Cylinder(1.2, 3.5)
print(cyl) # Cylinder(radius=1.2, height=3.5)

cyl = Cylinder(radius=1.2, height=3.5)
print(cyl) # Cylinder(radius=1.2, height=3.5)

cyl = Cylinder(height=3.5, radius=1.2)
print(cyl) # Cylinder(radius=1.2, height=3.5)

cyl = Cylinder(3.5, 1.2)
print(cyl) # Cylinder(radius=3.5, height=1.2)
```

Arguments *positionnels*: la position compte

Arguments *nommés*: variante plus lisible conseillée

Peuvent être réordrés!

Réordrer des arguments positionnels change la sémantique du code

Étape 2: avec classe + méthodes

```
@dataclass
class Cylinder:
    radius: float
    height: float

    def calc_volume(self) -> float:
        return math.pi * self.radius * self.radius * self.height

    def calc_surface_area(self) -> float:
        a1 = 2 * math.pi * self.radius * self.height
        a2 = 2 * math.pi * self.radius * self.radius
        return a1 + a2
```

Ces méthodes ont accès aux champs (sous-variables) stockées par l'objet (*radius* et *height*) pour faire leurs calculs via la référence à *self*, l'objet sur lequel elles ont été appelées. Ici, elle n'ont donc pas besoin de paramètres supplémentaires!

Les méthodes sont indentées car déclarées *dans* la classe

```
cyl = Cylinder(1.2, 3.5)
v = cyl.calc_volume()
a = cyl.calc_surface_area()
print(f"v = {v}, a = {a}")
```

Plus de fonctions «en vrac» dans le code; on appelle les méthodes que la classe *Cylinder* définit

Le paramètre *self* est fourni automatiquement pour l'appel de méthodes. Ici, il vaut *cyl*

Concepts principaux d'une classe

Une classe typique…

- modélise un objet (ou un concept abstrait) précis
- a un nom
- stocke des données avec ses champs (sous-variables)
- fournit des méthodes pratiques
  - Les méthodes ont toujours comme premier paramètre self, une référence à l'objet sur lequel la méthode est appelée
  - Les méthodes ont accès aux champs via le paramètre self

Self

- Le premier paramètre de chaque méthode est self
- Il n'est pas spécifié lors de l'appel, mais fourni automatiquement
  - Il référence toujours l'objet sur lequel la méthode est appelée
- Il sert à lire ou écrire des champs ou à appeler d'autres méthodes du même objet
- Par convention, on ne spécifie pas son type
- On peut toujours ajouter d'autres paramètres après si nécessaire pour que la méthode fasse son travail — comme on l'a appris pour les fonctions
  - Des valeurs pour ces autres paramètres doivent être fournies lors de l'appel

Résumé Cours 8

- Une classe définit un nouveau type
- Une classe déclare ses champs (sous-variables), peut définir des méthodes en plus
- Les méthodes sont liées à la classe et s'appellent «sur» une variable du type de la classe
- Pour chaque méthode, le paramètre automatique self est toujours à mentionner en premier lors de la définition de la méthode
  - Il est fourni automatiquement lors de l'appel
- Les méthodes ont accès aux champs (avec la notation self.xyz) et peuvent aussi changer leurs valeurs