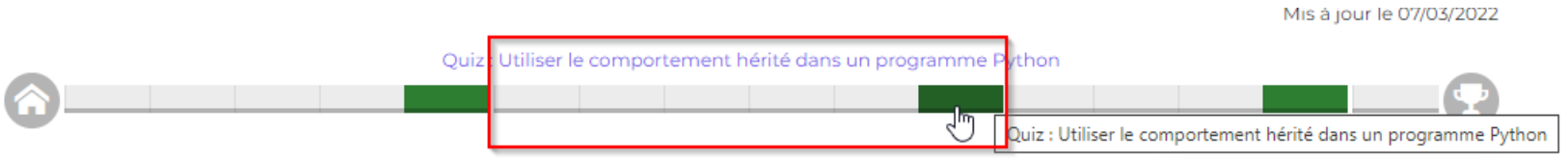
Exercice 01

```
SimplexSolver méthode build_tableau: <class 'list'>
SimplexSolver méthode solve: <class 'dict'>
Solution:
x1 = 2.0
x2 = 3.0
10) Affichez les coefficients de la fonction objective pour chaque variable X
coefficient de X1 = 3
coefficient de X2 = 2
11) Valeur de la fonction objective: 12.0 pour la solution X = ( 2.0,3.0)
12) Valeur de la fonction objective: 12.0 pour la solution X = ( 2.0,3.0)
13) Valeur des contraintes de la solution X = ( 2.0,3.0):
Contrainte 1: 2.0
Contrainte 2: 3.0
Contrainte 3: 12.0
```

Apprenez la programmation orientée objet en Python

Objectifs

1. Révision des concepts de classe, objet, méthode et attribut
2. Écrire une sous-classe
3. Surcharger une méthode
4. Utiliser une classe abstraite
5. Hiérarchie de classes
6. Utiliser une collection d’objets
7. Typer son code



1 - Révision - Quizz – POO

Compétence évaluée

- Ecrire des méthodes et des classes avec Python

2 - Héritage

Apprenez la programmation orientée objet en Python

Appliquez l'héritage dans votre code Python

Qu’est-ce que l’héritage?

L’héritage consiste à définir une sous-classe (ou classe enfant) à partir d’une classe parente de sorte que

- La sous-classe contient (ou hérite) des attributs/méthodes du parent.
- La sous classe peut aussi définir d’autres attributs/méthodes **spécifiques** à l’enfant.

Demo bibliothèque de films

Un système informatique gère le visionnement de films; lorsque l'on déclenche ce visionnement, on affiche le nom du film et on indique si nécessaire qu'il s'agit d'une cassette pour faire patienter l'utilisateur

```
Blade Runner va commencer...bon film!
Patience..., 2001: l'Odyssée de l'espace est assez long à rembobiner!
2001: l'Odyssée de l'espace va commencer...bon film!
```

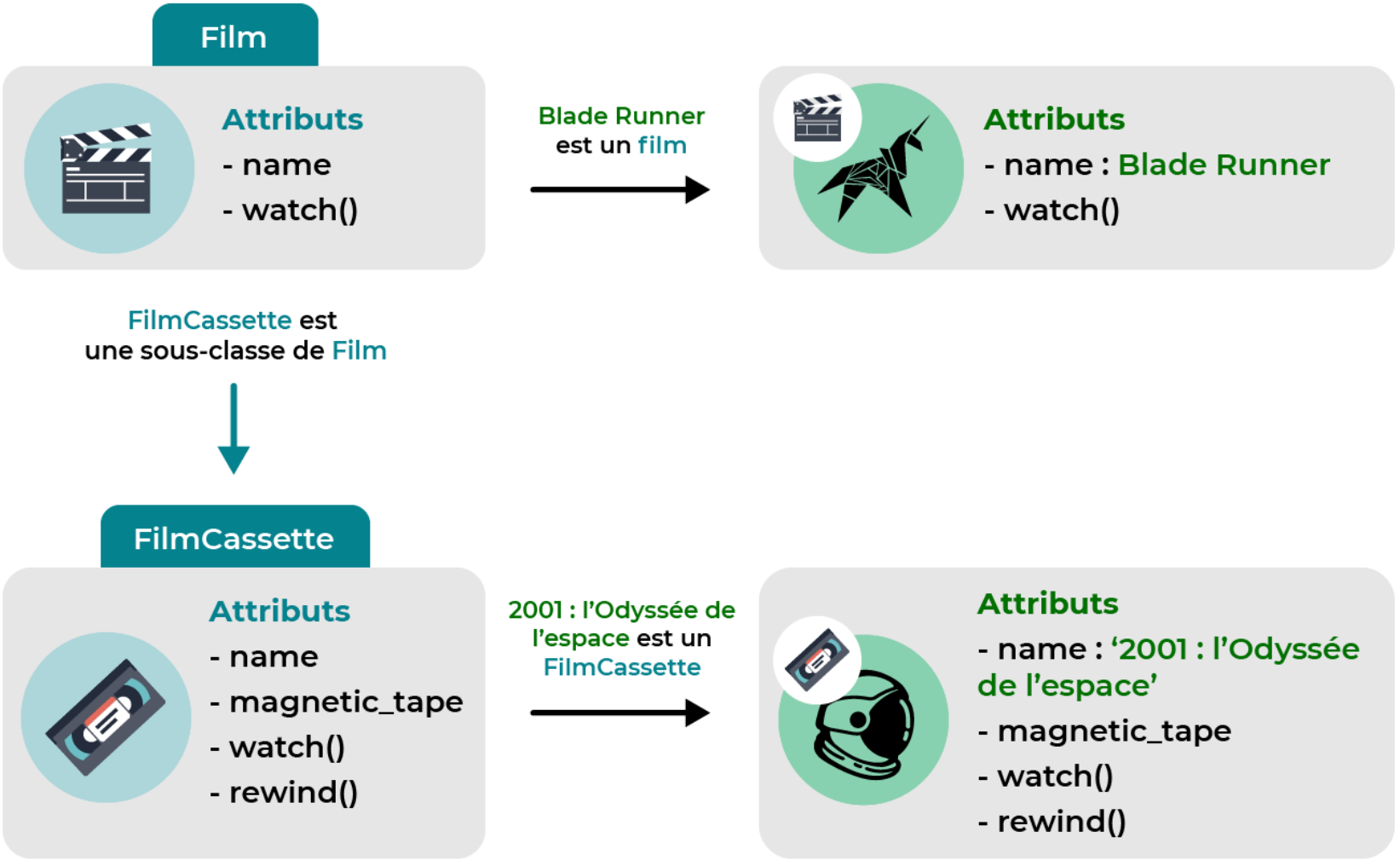
Demo bibliothèque de films

Votre code est responsable de l'affichage des messages;

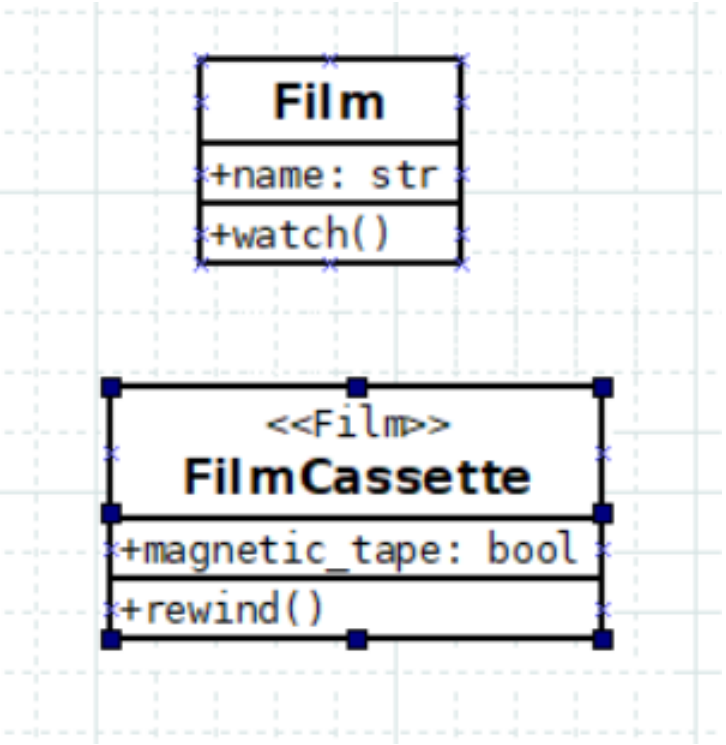
- Combien de classes?
- Héritage?
- Quels sont les attributs/méthodes du parent/enfant?

```
Blade Runner va commencer...bon film!
2001: l'Odyssée de l'espace va commencer...bon film!
Patience..., 2001: l'Odyssée de l'espace est assez long à rembobiner!
```

Demo Film


Film
Attributs
- name
- watch()
Blade Runner
est un film
Attributs
- name : Blade Runner
- watch()
FilmCassette est
une sous-classe de Film
FilmCassette
Attributs
- name
- magnetic_tape
- watch()
- rewind()
2001 : l'Odyssée de
l'espace est un
FilmCassette
Attributs
- name : '2001 : l'Odyssée
de l'espace'
- magnetic_tape
- watch()
- rewind()

Demo Film


Film
+name: str
+watch()
<<Film>>
FilmCassette
+magnetic_tape: bool
+rewind()

Pourquoi l’héritage?

- Réutiliser du code
- Ajouter des sous-classes sans modifier le code existant
- Modéliser un problème en séparant les responsabilités

Apprenez la programmation orientée objet en Python

Écrivez une sous-classe en Python

Syntaxe

```
class SousClasse(ParentUn,ParentDeux, ...)
```

Demo

```python
class FilmCassette(Film):
    """Un film en cassette !"""

    def __init__(self, name):
        """Initialise le nom et la bande magnetique."""
        self.name = name
        self.magnetic_tape = True

    def rewind(self):
        """Rembobine le film."""
        print("C'est long à rembobiner !")
        self.magnetic_tape = True
```

Résumé

- On définit une classe enfant en utilisant `class Enfant(Parent)` .
- Par défaut, toutes les classes héritent **d'Objet** – un objet Python qui fournit une fonctionnalité basique.
- Les classes peuvent hériter de classes parents multiples simultanément – dans le cas de l'**héritage multiple**.

__str__ versus __repr__

`__str__` is used in to show a string representation of your object **to be read easily** by others.

`__repr__` is used to show a string representation of **the** object.

Apprenez la programmation orientée objet en Python

Surchargez les méthodes en Python

Surcharge

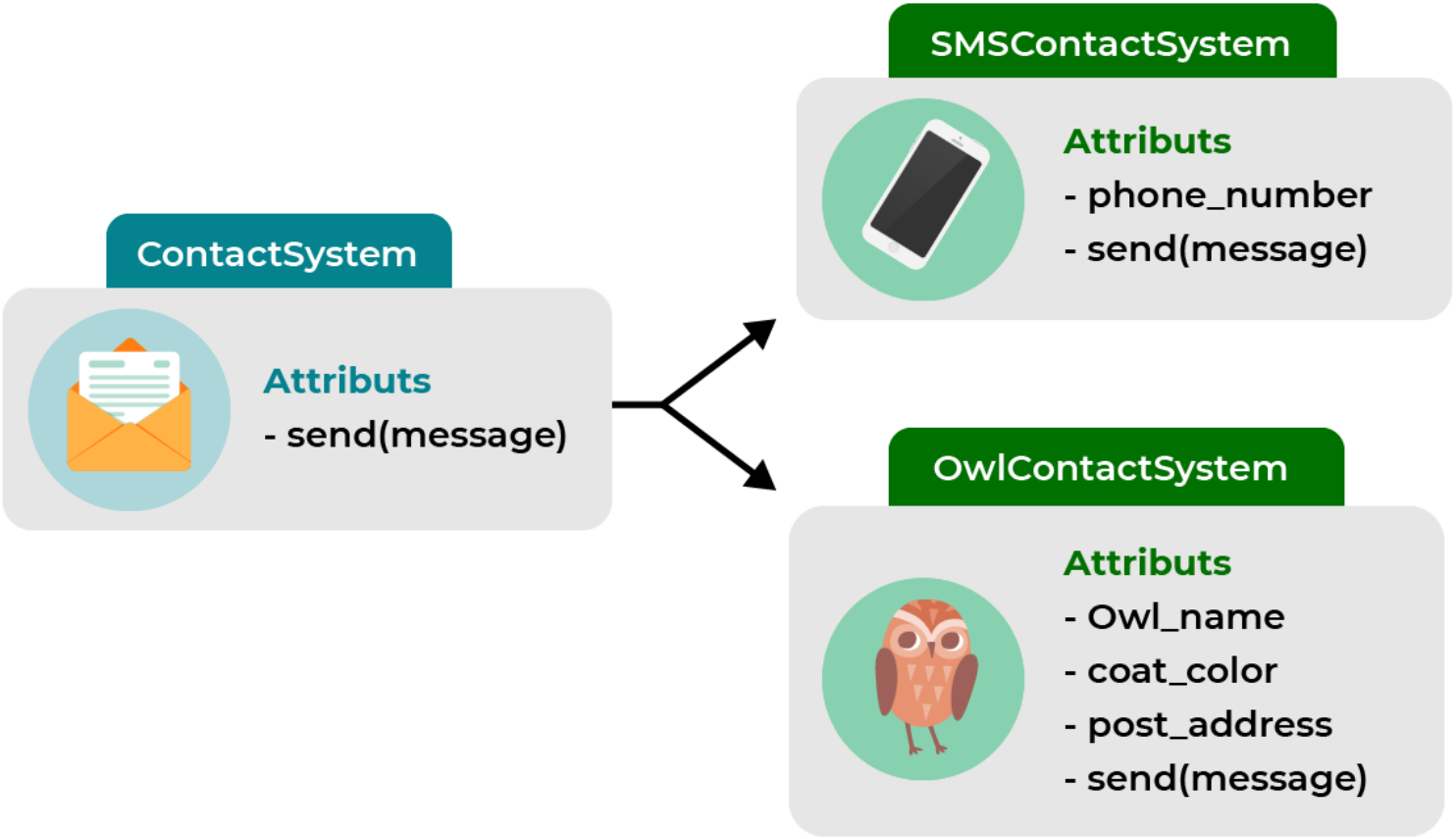
Signature d’une méthode

=

Nom + paramètres de la méthode

Lorsque une méthode définie dans une classe enfant utilise la même signature qu’une méthode de son parent, on dit qu’elle est surchargée ou redéfinie.

Quelle méthode est surchargée?



Réutilisez le code du parent

Lorsqu'un objet exécute une méthode surchargée, « le code de l'enfant l'emporte sur celui du parent ».

Pour accéder au code de la classe parente, utilisez la fonction super()

Demo bibliothèque de films

Pour améliorer notre système, on veut vérifier si le lecteur de film est en mesure de visionner des cassettes et réutiliser le plus de code possible.

```
5   unPlayer = Player("dvd")
6   blade = Film("Blade Runner")
7   blade.watch(unPlayer)
8   odyssee = FilmCassette("2001: l'Odyssée de l'espace")
9   odyssee.watch(unPlayer)
10  unSecondPlayer = Player("cassette")
11  odyssee.watch(unSecondPlayer)
```

```
Blade Runner va commencer...bon film!
Le lecteur n'est pas un lecteur de cassettes, le fim 2001: l'Odyssée de l'espace ne peut pas être lu!
Patience..., 2001: l'Odyssée de l'espace est assez long à rembobiner!
2001: l'Odyssée de l'espace va commencer...bon film!
```

```
@abstractmethod
def display(self):
    pass
```

Classe et méthode abstraites

- Une classe abstraite (Abstract Base Class) ayant une méthode abstraite, ne peut pas être instanciée.
- La seule façon de l'utiliser est de créer une sous-classe et d'implémenter la méthode abstraite.
- Cette technique permet de définir un « contrat » obligatoire pour des classes

```
@abstractmethod
def display(self):
    pass
```

Classe et méthode abstraites

```
#importer le module ABC et l'annotation abstractmethod
from abc import ABC, abstractmethod

#MaClasse est abstraite car elle hérite de ABC
class MaClasse(ABC):
        #doSomething est une méthode abstraite
        #qui doit être surchargée par ses enfants
        @abstractmethod
        def doSomething(self):
            pass
```

Démo Film

ClasseAbstraite\film.py

```python
from abc import ABC, abstractmethod


class Film(ABC):
    def __init__(self, name):
        self.name = name
    @abstractmethod
    def watch(self, player):
        print(f"{self.name} va commencer...bon film!")
```

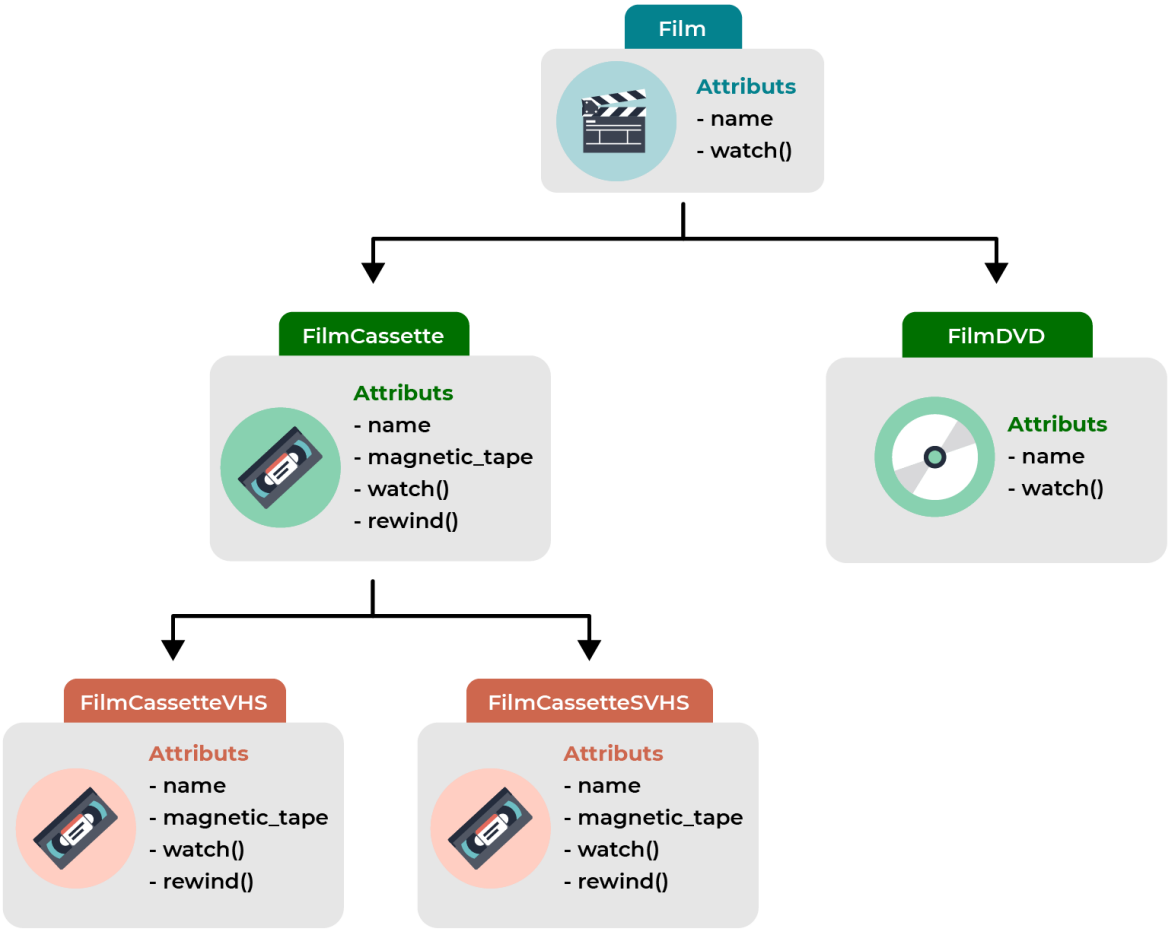
Résumé

- Une classe enfant peut fournir **sa propre implémentation** d'un élément hérité de sa classe parent.
- L'implémentation de la classe enfant est prioritaire sur celle du parent – elle **surcharge** l'implémentation de la classe parent.
- On peut utiliser la méthode **super** pour accéder à des méthodes dans la classe parent que nous avons surchargée.
- Une **classe abstraite** est une classe qui **ne peut pas être instanciée** – à la place, il faut en hériter.

Apprenez la programmation orientée objet en Python

Utilisez une hiérarchie d'héritage et l'héritage multiple

Exemple – hiérarchie d'héritage



Method Resolution Order

L’héritage multiple n’est pas une bonne pratique (problème du diamant)
Si 2 parents implémentent la même méthode, Python exécutera celle implémenté en premier (MRO)

Résumé

- Une classe parent peut elle-même avoir un parent.
- Lorsque nous avons de multiples niveaux d'héritage, nous avons une **hiérarchie d'héritage**.
- Une classe peut aussi hériter de multiples classes parents dans le cadre de ce que l'on nomme l'**héritage multiple**.

Apprenez la programmation orientée objet en Python

Utilisez des objets dans des collections

Collection

```
1 numberList = [1.2, 3.5, 4.3, 2.8]
2 phoneNumberDictionary = { "alice" : "01234 567890", "bob" : "01324 765098"}
3 mixedList = [5, 4, 3, 2, 1, "boom"]
```

Les collections comprennent
des **listes** – où les données ont une position
et sont **indexables**
et des **dictionnaires** – où on attribue
une **clé** aux données.

Dictionnaire

- Structure de données sous forme de liste de paire {clef:valeurs} (proche de JSON)
- La clef est unique, de type immutable
- Démo; créer, accéder, ajouter/supprimer, itérer

Objet immutable

Les clefs de dictionnaires doivent être des valeurs **immutables** (“immuables”), valeur qui ne peuvent être changée après sa création comme des nombres et des chaînes de caractères

Par défaut, toutes les classes que vous écrirez seront **mutables** (“muables”)

Duck Typing

```
1 volunteers = [Person("Alice"), Fish("Wanda"), Person("Bob")]
2
3 for volunteer in volunteers:
4     volunteer.walk()  # Oops!
```

Les poissons ne peuvent pas marcher, donc si nous exécutons ce code, nous obtiendrons ce genre de chose : `AttributeError: 'Fish' object has no attribute 'walk'` . 🐠🐠

Documentez votre code

```python
class Cat:
    """Un chat."""

    def meow(self):
        """Miaule."""
        print("Meow!")

    def say(self, to_say):
        print(f"say de Cat {to_say}")
```

Typez votre code

python

```python
from typing import List


def highest(numbers: List[int]) -> int:
    max_value = 0
    for number in numbers:
        if number > max_value:
            max_value = number
    return max_value
```

Les IDE prenant en compte des bibliothèques de vérification de types (telles que mypy) pourront vous avertir d'un typage non respecté. L'avantage du typage réside aussi dans **l'autocomplétion** fournie par votre IDE. En effet, typer les paramètres de fonction, par exemple, permettra à votre IDE de proposer tous les attributs disponibles pour ce paramètre. 🤗

Programmation défensive

IF

```python
def divide(a, b):
    if b != 0:
        return a / b
    else:
        return 0
```

ASSERT (pas en production)

```python
def divide(a, b):
    assert b != 0
    return a / b
```

ISINSTANCE() HASATTR()

```python
for volunteer in volunteers:
    if hasattr(volunteer, "walk"):
        volunteer.walk()

if not isinstance(x, Y):
    raise Z("M")
```

Résumé

- Les objets peuvent être stockés dans des collections – tout comme n'importe quoi d'autre.
- Python gère les types avec ce que l'on appelle le **duck typing** – si ça a un bec et cancane comme un canard, c'est probablement un canard.
- Nous devons être particulièrement attentifs, lorsque nous utilisons les collections, à ne pas essayer d'accéder à des attributs qu'un objet ne possède pas.