

DAWAN Paris
DAWAN Nantes
DAWAN Lyon

11, rue Antoine Bourdelle, 75015 PARIS
32, Bd Vincent Gâche, 5e étage - 44200 NANTES
Bt Banque Rhône Alpes, 2ème étage - 235 cours Lafayette 69006 LYON



Formation Python: Programmation Orientée Objet

Plus d'info sur <http://www.dawan.fr> ou 0810.001.917

Formateur: Matthieu LAMAMRA

Python Orienté Objet

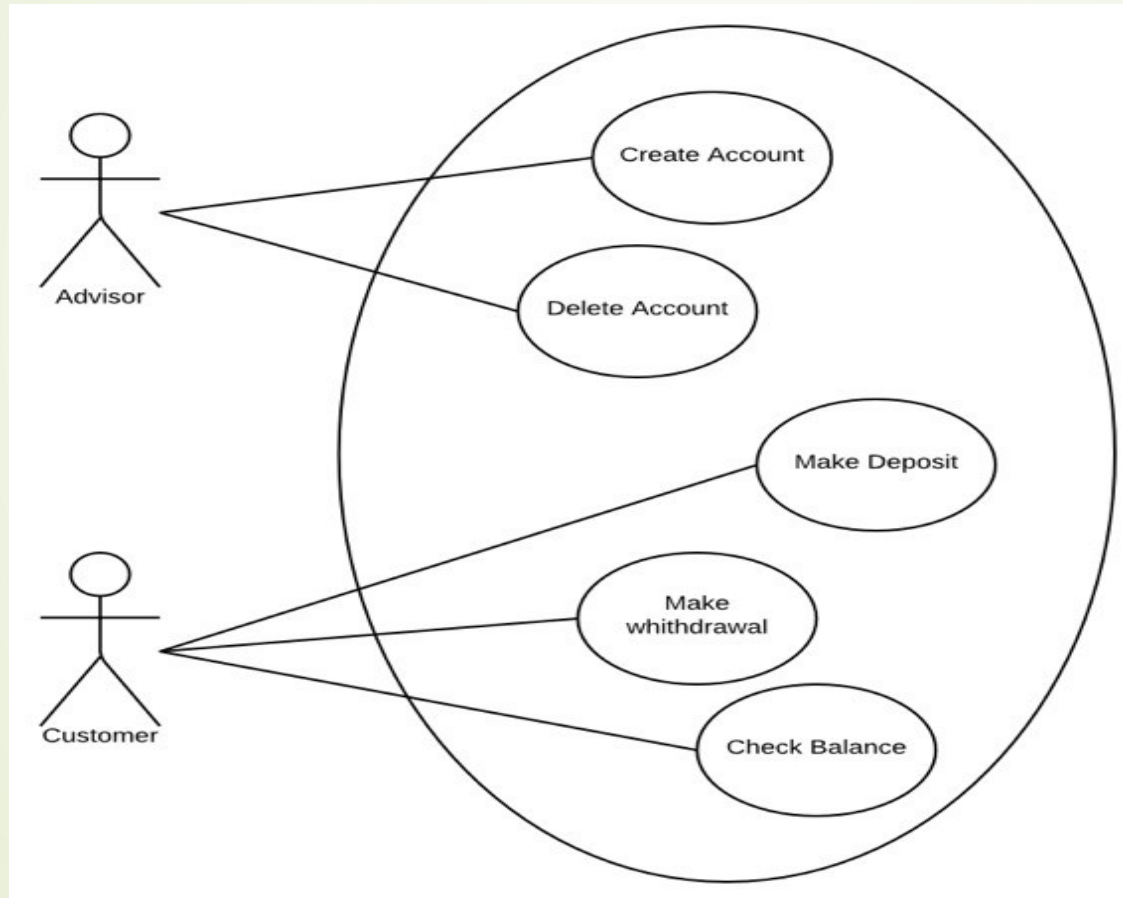
- Paradigmes de programmation
 - Modèles de pensée et d'écriture de code comme chemin du problème vers la solution
 - Programmation **impérative** : **blocs d'instructions** => machine d'état
 - Programmation **fonctionnelle** : **fonctions et valeurs** => $S = f(P)$
 - Programmation orientée objet « **POO** » : **classes et d'objets** => assemblage de « légos »
 - Les classes sont des définitions de **structures** comprenant :
 - des variables appelées **attributs**
 - des fonctions appelées **méthodes**
 - Les objets sont des **instances** ou des réalisations de la classe, comme un ouvrage d'art l'est d'un plan => le **type** d'un objet est sa classe

Python Orienté Objet

- UML : Unified Modeling Language
 - Langage de **modélisation graphique standardisé** : 14 types de diagrammes en 2.5
 - Répartis sur 3 vues :
 - **Fonctionnelle** : acteurs et besoins
 - diagrammes de cas d'utilisation
 - **Statique** : les structures de la solution et leur relations de dépendance
 - diagrammes de classe
 - diagrammes d'objet
 - **Dynamique** : comportement de ces structures dans le temps
 - diagrammes de séquence

Python Orienté Objet

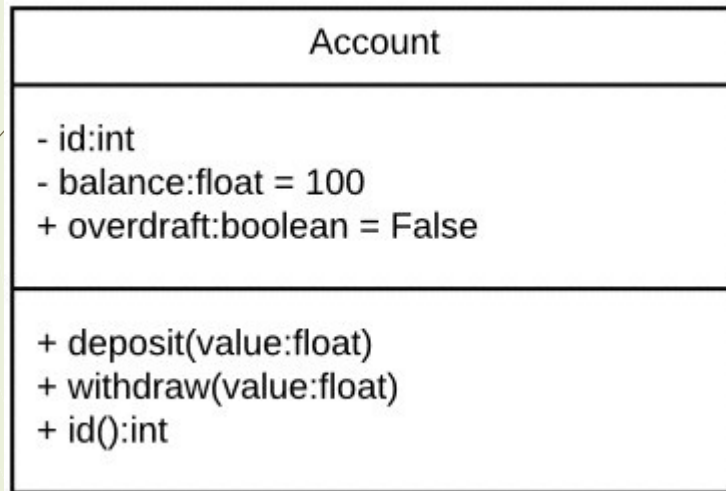
- UML : Diagrammes de cas d'utilisation



Python Orienté Objet

- Classes : Représentation et déclaration

➤ Diagramme de classe UML



ATTRIBUTS

METHODES

/ Création de classe Python

```
class Account:
    id = 0
    balance = 100.0
    overdraft = False

    def deposit(self, amount):
        self.balance += amount

    def get_id(self):
        return self.id
```

self est la référence d'instance : cette variable référence l'objet instancié
Une méthode déclare toujours **self comme premier argument**

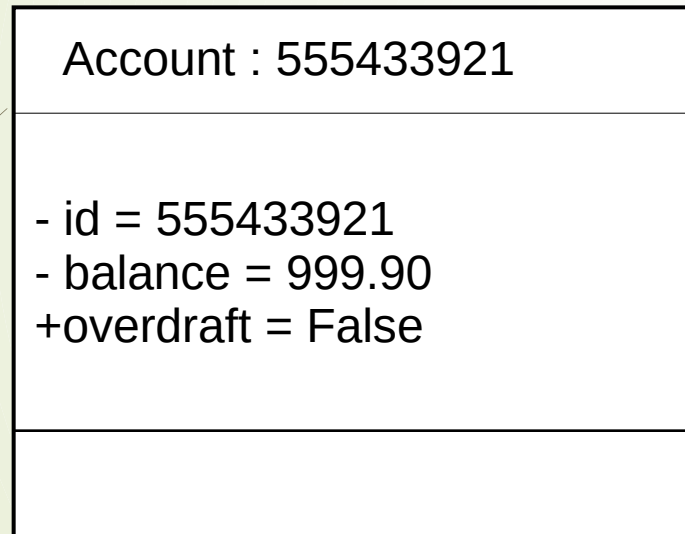
Python Orienté Objet

- Objets : Représentation et instanciation

➤ Diagramme d'objet UML

/

Instanciation d'objet Python



```
account = Account()
```

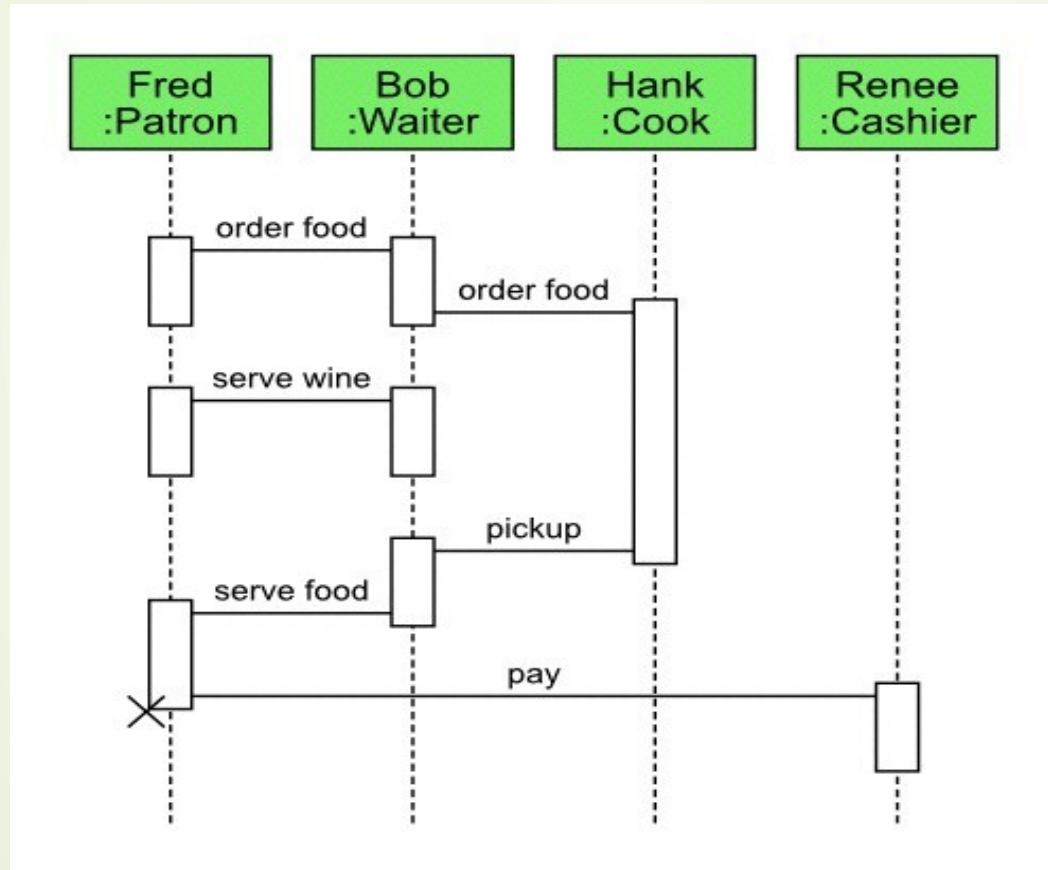
```
account.id = 555433921  
account.balance = 999.90  
account.overdraft = False
```

```
account.get_id()  
account.deposit(0.10)
```

- comme les fonctions, les classes sont des objets « **callables** »
- instancier consiste à appeler la classe
- La variable interne **self** représente account
- **On ne place pas self à l'appel des méthodes**

Python Orienté Objet

- UML : Diagrammes de séquence



Python Orienté Objet

- Visibilité des attributs et méthodes
 - En python, par défaut l'accès aux attributs et méthodes est **public** : on peut les afficher et les affecter directement via l'opérateur « . » à l'extérieur de l'objet
 - Un attribut ou une méthode précédé de deux underscores « __ » est **privé**, i.e **inaccessible** depuis l'extérieur de l'objet (ou presque), et **absent de la documentation**
 - On s'astreint à n'y accéder que par une **méthode publique => c'est l'encapsulation**
 - un attribut précédé d'un underscore « _ » est **public**, mais n'apparaît pas dans la documentation de la classe
 - Une **méthode privée** n'est appellable que depuis le corps d'une autre méthode

Python Orienté Objet

- Visibilité des attributs et méthodes

```
class Account:
    __balance = 100.0

    def __update_balance(self, amount):
        self.__balance += amount

    def deposit(self, amount):
        if amount > 0:
            self.__update_balance(amount)

    def withdraw(self, amount):
        if amount > 0:
            self.__update_balance(-amount)

    def get_balance(self):
        return self.__balance
```

```
acc = Account()

# AttributeError
# acc.__balance
# acc.__update_balance(200)

acc.deposit(200)
acc.withdraw(100)
acc.get_balance()
```

Python Orienté Objet

- Docstrings, Introspection
 - Les **docstrings** sont disponibles pour documenter les classes, et nourrir la fonction **help()**
 - La fonction **isinstance(obj, ClassName)** teste si l'objet obj est de classe ClassName
 - La fonction **dir(obj)** affiche tous les attributs et méthodes de l'objet obj
 - Parmi ces méthodes on trouve :
 - **__init__** : initialise l'objet, exécutée à **l'instanciation**
 - **__del__** : appelée avant la destruction de l'objet par le mot clé **del**
 - **__str__** : permet de convertir l'objet en chaîne de caractère, appelée par **str()** et **print()**

Python Orienté Objet

- « initialiseur »

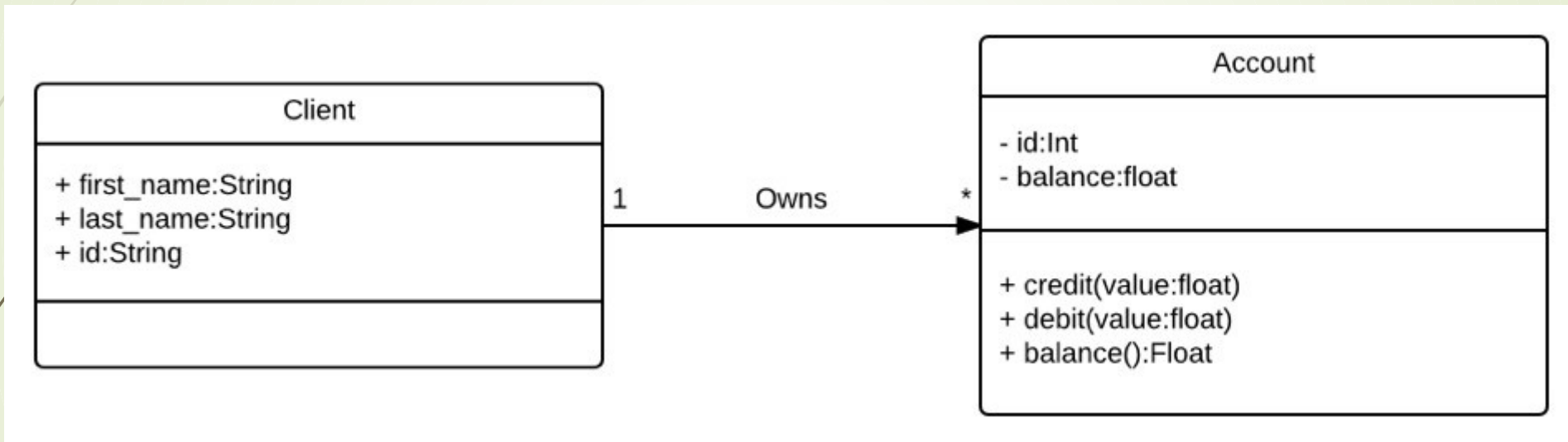
- En redéfinissant « l'initialiseur » `__init__`, on peut initialiser l'objet dès l'instanciation
- On peut créer des attributs dans `__init__` (bonne pratique) ou toute autre méthode
 - on parle alors d'**attributs d'objet**, qui n'appartiennent qu'à l'objet instancié
- Les attributs créés précédemment sont appelés **attributs de classes**
 - ils existent pour la classe et sont transférés comme attributs d'objet à l'instanciation

```
class Account:  
  
    def __init__(self, balance=100.0):  
        self.balance = balance  
  
    def deposit(self, amount):  
        self.balance += amount
```

```
acc = Account(200)  
  
acc.deposit(200)  
  
acc.balance  
# 400.
```

Python Orienté Objet

- Associations entre classes : UML



- Un client peut posséder n comptes : « * » ou « min..max »
- Un compte n'est possédé que par 1 client
- **l'aggrégation** est une association de type « ensemble <=> partie »
- **La composition** est une aggrégation de type « ensemble <=> partie essentielle et dépendante »

Python Orienté Objet

- Associations entre classes : Python

```
class Client:
    def __init__(self, first, name):
        self.first = first
        self.name = name

    def get_full_name(self):
        return f"{self.first.capitalize()} {self.name.upper()}"

class Account:
    def __init__(self, balance, client):
        self.balance = balance
        self.client = client

    def get_client_name(self):
        return self.client.get_full_name()
```

```
cl = Client("jean", "dupont")
```

```
acc = Account(200, cl)
```

```
acc.get_client_name()
# Jean DUPONT
```

Account ne connaît que les méthodes publiques de Client
=> Couplage faible entre les classes : c'est l'injection de dépendance

Python Orienté Objet

- En Python, tout est objet !!!

```
x = -2  
  
x.__abs__()  
2  
  
isinstance(x, int)  
True  
  
"bonjour".upper()  
'BONJOUR'  
  
isinstance("bonjour", str)  
True
```

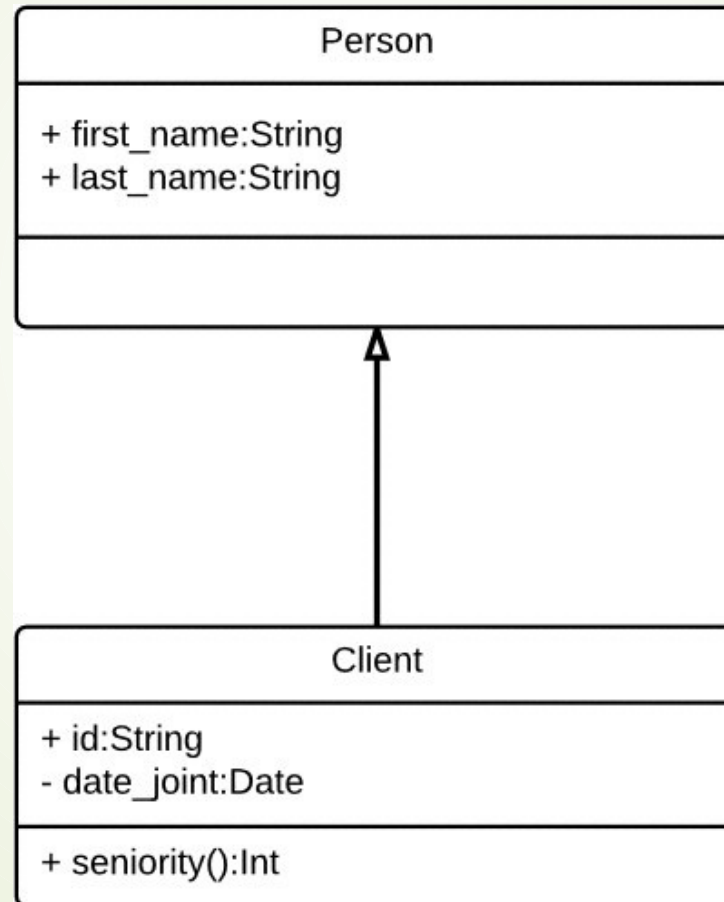
```
isinstance([1, 2], list)  
True  
  
class Test():  
    pass  
  
t = Test()  
  
type(t)  
__main__.Test
```

Python Orienté Objet

- L'héritage
 - L'héritage décrit une relation « général \Leftrightarrow particulier » entre deux classes A et B
 - La classe fille B a accès aux mêmes attributs et méthodes que la classe mère A
 - La classe fille B peut rajouter des attributs et méthodes propres
 - La classe fille B peut redéfinir ou surcharger une méthode héritée de sa mère

Python Orienté Objet

- L'héritage : UML



Python Orienté Objet

- L'héritage : Python

```
class Person:
    def __init__(self, first, name):
        self.first = first
        self.name = name

    def get_full_name(self):
        return f"{self.first.capitalize()} {self.name.upper()}"

class Client(Person):
    def __init__(self, _id, first, name):
        super().__init__(first, name)
        self.__id = _id

    def get_id(self):
        return self.__id
```

```
# __init__ : méthode surchargée
cl = Client( 42, "jean", "dupont")
```

```
# méthode héritée
cl.get_full_name()
```

```
# méthode propre
cl.get_id()
```

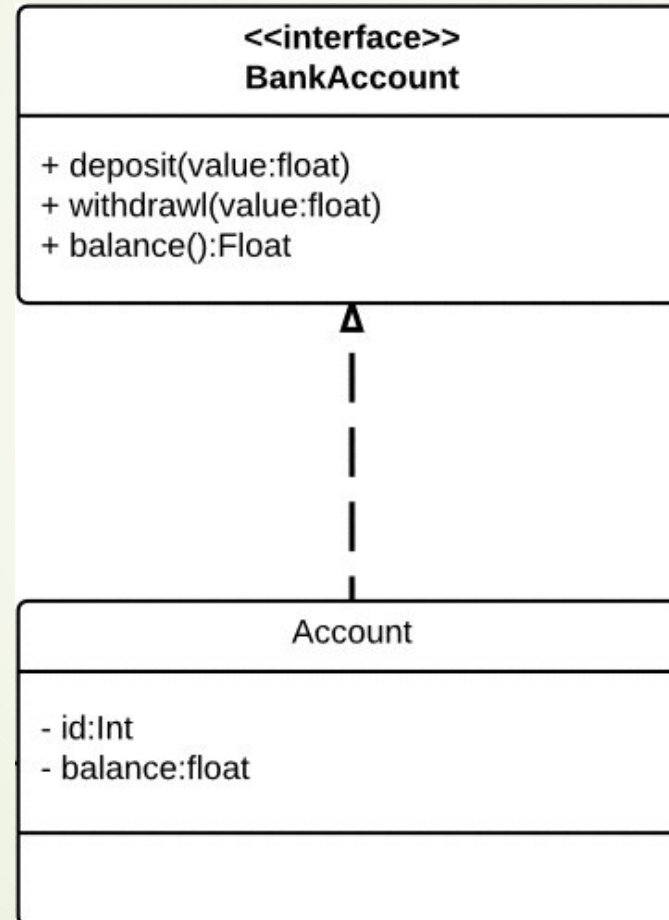
super() appelle une méthode de la classe mère sur l'objet courant (ici cl)

Python Orienté Objet

- Les « interfaces »
 - En POO, une classe est **abstraite** si elle n'est **jamais instanciée**, mais simplement héritée
 - En python, une **interface** est semblable à une classe abstraite n'ayant que des méthodes
 - Ces méthodes peuvent même être vides => **elles assurent la signature** des méthodes filles

Python Orienté Objet

- Les « interfaces »



Python Orienté Objet

- Créer des objets itérables

- Itérables : classes qui implémentent les méthodes implicites `__init__`, `__iter__` et `__next__`
- Les objets instanciés peuvent être utilisés avec `iter()` et `next()`, et dans les boucles `for`

```
class It:
    def __init__(self, lim=3):
        self.lim = lim

    def __iter__(self, cpt=0):
        self.cpt = cpt
        return self

    def __next__(self):
        if self.cpt < self.lim:
            ret = self.cpt
            self.cpt += 1
            return ret
        else: raise StopIteration
```

```
obj = It()

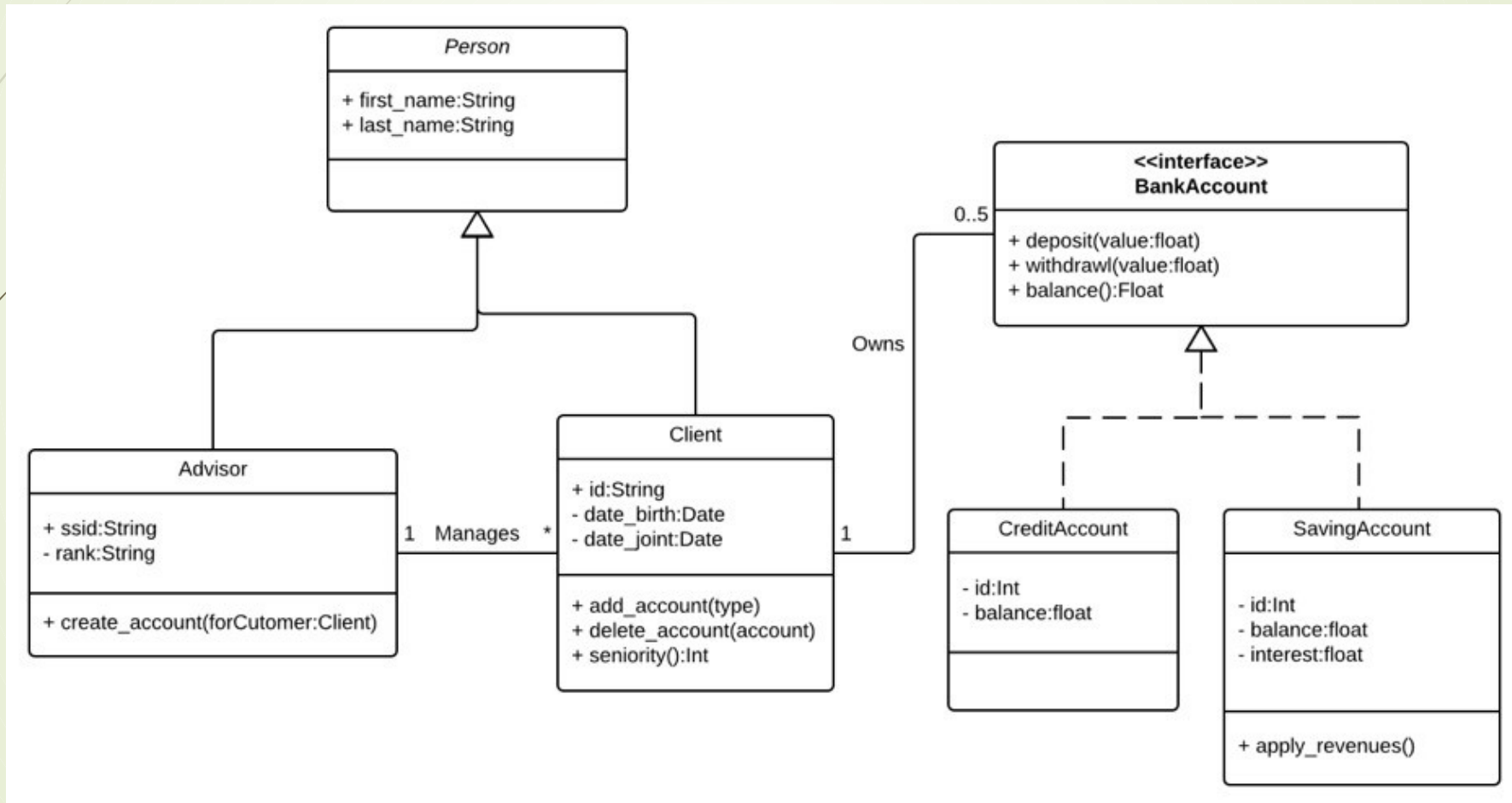
It = iter(obj)

next(it) # 0
next(it) # 1
next(it) # 2
next(it) # StopIteration

for elem in it:
    print(elem)
```

Python Orienté Objet

- UML : exemple complet



Les exceptions python

- Principe

- Mécanisme d'**interruption de l'exécution** du programme et **signal d'une erreur** d'exécution
- Retourne un **objet Exception**
- Peut être **gérée** par le programmeur, ou interrompt le programme et affiche la « **stack trace** »

Les exceptions python

- Exception non gérées

```
In [1]: import nimportequoi
-----
ModuleNotFoundError                                Traceback (most recent call last)
<ipython-input-1-a79b241ddd83> in <module>
----> 1 import nimportequoi

ModuleNotFoundError: No module named 'nimportequoi'

In [2]: dct = {"key": "value"}

In [3]: dct["nope"]
-----
KeyError                                Traceback (most recent call last)
<ipython-input-3-7a4844c78e2f> in <module>
----> 1 dct["nope"]

KeyError: 'nope'
```


Les exceptions python

- Gestion des exceptions
 - Délimiter le code pouvant lever une exception par les instructions **try et except**
 - Si une **exception est levée** dans le bloc try, le bloc except est appelé
 - Le bloc except doit **déclarer les exceptions gérées** (Bonne pratique)

```
In [1]: currencies = ["Dollar", "Euro", "Yuan", "Yen"]

In [2]: try:
...:     for i in range(5):
...:         cur = currencies[i]
...: except IndexError:
...:     print("pas assez de monnaies")
...:
pas assez de monnaies
```

Les exceptions python

- Gestion des exceptions multiples
 - On peut multiplier les blocs except pour prévenir plusieurs types d'erreurs

```
In [1]: currencies = ["Dollar", "Euro", "Yuan", "Yen"]

In [2]: try:
...:     for i in range(5):
...:         cur = currencies[i]
...:         price = float(cur)
...: except IndexError:
...:     print("pas assez de monnaies")
...: except ValueError:
...:     print("{} : {} ne peut pas être converti".format(cur, type(cur)))
...:
Dollar : <class 'str'> ne peut pas être converti
```

Les exceptions python

- La mère de toutes les exceptions
 - L'exception de classe **Exception** est héritée par toutes les autres : elle intercepte tout

```
In [3]: try:
...:     for i in range(5):
...:         cur = currencies[i]
...:         price = float(cur)
...: except Exception:
...:     print("problem")
...:
problem
```

Les exceptions python

- Else et finally

- **else** permet d'exécuter du code uniquement lorsqu' **aucune exception n' a été levée**
- **finally** est exécuté après tous les autres blocs **qu'il y ait eu exception ou non**

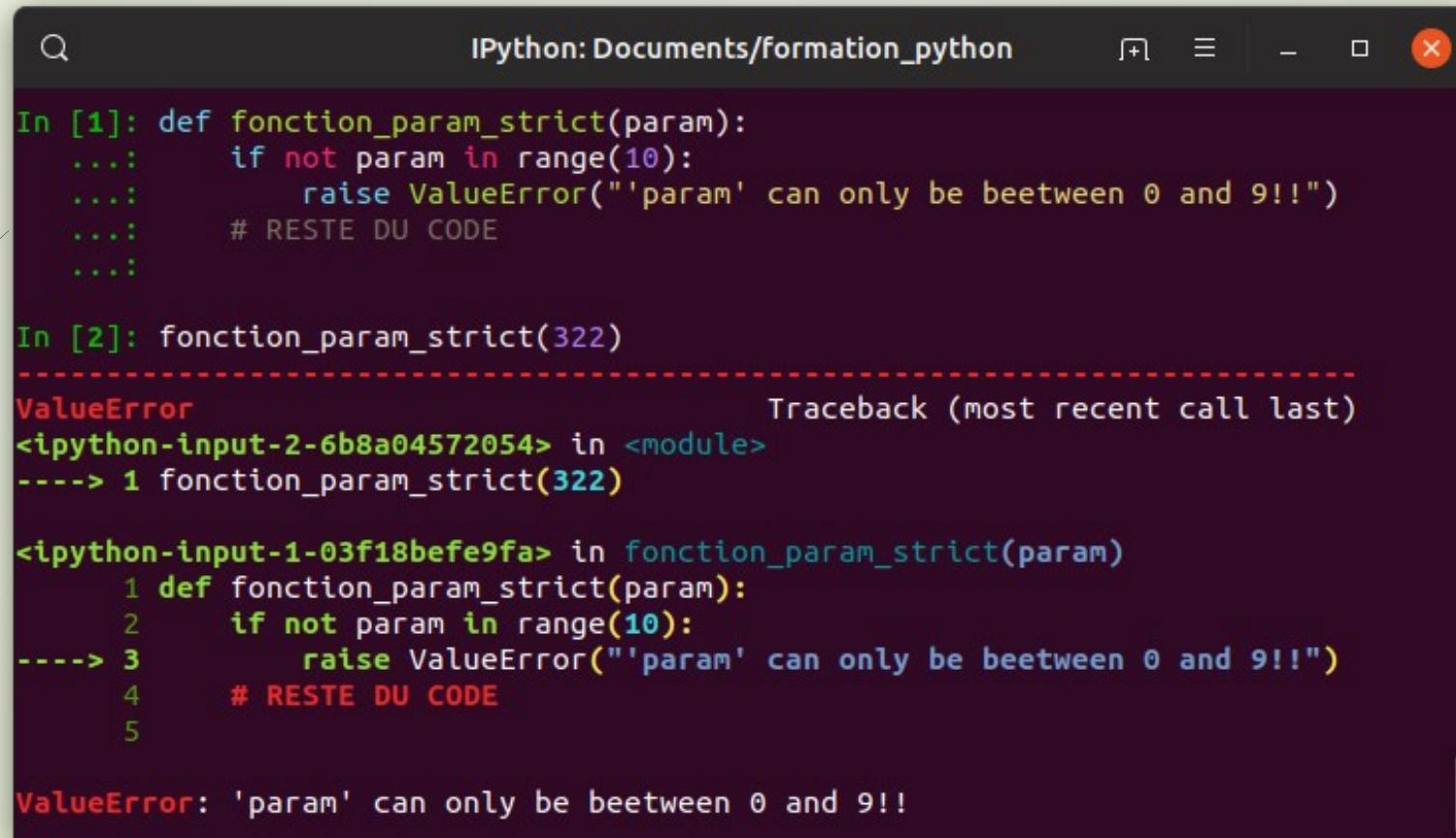
```
def getCurrency(locale):  
    currencies = {"fr": "€", "us": "$", "uk": "£"}  
    return currencies[locale]
```

```
In [6]: try:  
...:     french_price = "30.99 " + getCurrency("fr")  
...:     brasilian_price = "519 " + getCurrency("br")  
...: except KeyError as e:  
...:     print(e)  
...: else:  
...:     print("prix renseignés !!")  
...:  
...:  
...:  
'br'
```

```
In [8]: try:  
...:     french_price = "30.99 " + getCurrency("fr")  
...:     american_price = "34 " + getCurrency("us")  
...: except KeyError as e:  
...:     print(e)  
...: else:  
...:     print("prix renseignés !!")  
...:  
prix renseignés !!
```

Les exceptions python

- Déclencher une exception
 - Une exception peut être déclenchée par le programmeur grâce à l'**Instruction raise**



```
IPython: Documents/formation_python

In [1]: def fonction_param_strict(param):
...:     if not param in range(10):
...:         raise ValueError("'param' can only be between 0 and 9!!")
...:     # RESTE DU CODE
...:

In [2]: fonction_param_strict(322)

-----
ValueError                                Traceback (most recent call last)
<ipython-input-2-6b8a04572054> in <module>
----> 1 fonction_param_strict(322)

<ipython-input-1-03f18befe9fa> in fonction_param_strict(param)
      1 def fonction_param_strict(param):
      2     if not param in range(10):
----> 3         raise ValueError("'param' can only be between 0 and 9!!")
      4     # RESTE DU CODE
      5

ValueError: 'param' can only be between 0 and 9!!
```