PYTHON, PROGRAMMATION OBJET

PYTHON : PLAN DE LA FORMATION

- Introduction
- La boite à outils Python
- Les bases
- La programmation orientée objet
- La stdlib
- Qualité

PYTHON :

INTRODUCTION

IT WAS NICE TO LEARN PYTHON;
A NICE AFTERNOON

D. Knuth, Trento, 2012

HISTORIQUE

- Créé en 1989 par Guido Van Rossum
- 1991 : première version publique (0.9.0)
- 2001 : Fondation Python
- 2008 : Python 3
- 2005 : Guido Van Rossum rejoint Google
- 2012 : Guido Van Rossum rejoint Dropbox

QUI UTILISE PYTHON ?

https://wiki.python.org/moin/OrganizationsUsingPython

- Google (Google spider, search engine, Google app engine), YouTube
- Red Hat (Anaconda)
- Walt Disney Feature Animation et ILM
- NASA
- Instagram, Pinterest
- EVE Online

CARACTÉRISTIQUES DU LANGAGE

- Open Source
- Langage interprété
- Multiplate-formes
- Multi-paradigmes
- Haut niveau
- 2 fois « programming language of the year » TIOBE (2007 et 2010)

LANGAGE INTERPRÉTÉ

- Pas de phase de compilation
- Moins performant qu'un langage compilé

LANGAGE INTERPRÉTÉ

- CPython : interpréteur de référence en C
- Autres interpréteurs : Jython (Java), IronPython (.Net)
- Performance : **PyPy** (Attention, certaines incompatibilités)

CARACTÉRISTIQUES DU LANGAGE

- Langage de haut niveau
- Pas de gestion de mémoire
- Typage dynamique
- Fortement typé

PYTHON 2 OU PYTHON 3 ?

- Python 2.7
  - Existant encore fortement en Python 2.x
  - Existe encore des dépendances en Python 2.x
- Nouveau projet devrait utiliser Python 3
- Transition facilitée par le package future

PYTHON :

BOITE À OUTILS

LES OUTILS NÉCESSAIRES

- Interpéteur
- Gestionnaire de packages
- iPython
- Environnement de développement

OUTIL INDISPENSABLE : L'INTERPRÉTEUR

- Installation
  https://www.python.org/downloads/
- Linux : gestionnaires de paquets
- Os X : Python 2.7 installé par défaut

L'INTERPRÉTEUR INTERACTIF

- Dans un terminal, lancer Python par la commande python

LE HELLO WORLD

```
>>> print("Hello World")
Hello World
```

GESTIONNAIRE DE PACKAGES : PIP

- Installé dans certains cas
- Installation, voir : https://pip.pypa.io/en/stable/installing/
- Mise à jour : pip install -U pip
- Mise à jour Windows : python -m pip install -U pip
- Informations : https://pypi.python.org/pypi/pip

GESTIONNAIRE DE PACKAGES : PIP

- Usage :
  - sudo pip install package
  - pip install --user package
  - pip install package --upgrade
  - pip uninstall package

IPYTHON

- Interpréteur interactif évolué
- Installation : pip install ipython
- Informations : http://ipython.org/

ENVIRONNEMENT DE DÉVELOPPEMENT

Assurez-vous d'être en UTF-8

- Eclipse + PyDev
  http://www.pydev.org/

- PyCharm (existe en Commercial ou Community)
  https://www.jetbrains.com/pycharm/download/

PORTABILITÉ PYTHON 3

Pour assurer la portabilité du code Python 2 vers Python 3, utiliser le package future

- http://python-future.org/
- pip install future

PYTHON : MANIPULER DES DONNÉES

EN PYTHON, TOUT EST OBJET

- Les objets peuvent avoir des attributs
  monObjet.monAttribut

- Les objets peuvent avoir des méthodes
  monObjet.maMethode()

LES VARIABLES

- Contiennent une référence
- Type déterminé dynamiquement
- Information sur le type avec la fonction type()
- Détruites lorsqu'elles ne sont plus accessibles
- Règles nommage : peut contenir a-z, A-Z, _ ou 0-9 (sauf premier caractère) ([a-zA-Z_][a-zA-Z0-9_]*)
- Sensibles à la casse

CONVENTIONS CODAGE

- Définies dans la PEP8 : https://www.python.org/dev/peps/pep-0008/
- Lettres seules, en minuscule : pour les boucles et les indices
- Lettres minuscules + underscores : pour les modules, variables, fonctions et méthodes
- Lettres capitales + underscores : pour les (pseudo) constantes
- Camel case : nom de classe

LES VARIABLES : EXEMPLES

```
>>> answer = 42
>>> PI = 3.14
>>> hello = "Hello World"
>>> stuff = [42, "Hello World", 3.14]
>>> days_of_week = ('Lundi', 'Mardi', 'Mercredi')
>>> dic = {'Lundi': 1, 'Mardi': 2, 'Mercredi': 3}
>>> nothing = None
>>> type(PI)
<type 'float'>
```

LES VARIABLES : AFFECTATION

```
>>> var = 42
>>> var_1 = var_2 = 42
>>> var_1, var_2 = "Hello World", 42
>>> (var_1, var_2, var_3) = [42, "Hello World", 3.14]
```

COMPRENDRE LES VARIABLES

- Afficher la variable : print(var)
- Afficher le type : print(type(var))
- Afficher la classe : print(var.__class__)
- Afficher le nom de la classe : print(var.__class__.__name__)
- Afficher les méthodes d'un objet : print(dir(var))
- Afficher l'aide sur l'objet : help(var)

PRINT : COMPATIBILITÉ PYTHON 3 - PEP 3105

- En Python 3, print est une fonction
- En Python 2, print 'Hello World' est autorisé
- print('Hello', 'World') retourne un tuple en Python 2 et une chaine de caractères en Python 3
- Portabilité assurée par l'instruction
  from __future__ import print_function

TYPES NUMÉRIQUES

- 4 types numériques : int, long, float et complex
- Les booléens (True et False) sont des sous-types des entiers int
- Il est possible d'imposer le type par les constructeurs int(), long(), float() et complex()

TYPES NUMÉRIQUES, ENTIERS ET LONGS

- Deux types pour les entiers : int et ~~long~~
- Python 3 ne déclare plus que des int
- Valeur d'un int est comprise entre -2^(n-1) et 2^(n-1)-1
- Valeur maximum d'un int donnée par sys.maxint
  sys.maxsize (taille maximum de conteneurs en Python)
- Un int correspond à une variable manipulée dans un registre

EXEMPLE DE DÉCLARATION DE TYPES

```
>>> var_int = int(42.2)
>>> var_int = 42

>>> var_float = float(3)
>>> var_float = 3.14

>>> var_long = long(42)
>>> var_long = 42L

>>> var_cpx = complex(42, 2)
>>> var_cpx = 42+2j
>>> var_cpx.real
42.0
>>> var_cpx.imag
2.0
```

TYPES NUMÉRIQUES, ÉCRITURE LITTÉRALE

- binaire : 0b01010101
- octal : 0755 ou 0o755
- hexadécimal : 0x41
- puissance de 10 : 3.14e-10

OPÉRATEURS SUR LES TYPES NUMÉRIQUES

x+y	Addition
x-y	Soustraction
x*y	Multiplication
x/y	Division
x//y	Division entière
x%y	Reste
-x	Opposé
+x	
x**y	Puissance

DIVISION : COMPATIBILITÉ PYTHON 3 - PEP 238

- En Python 2
  - La division de int ou long est une division entière
  - La division de float donne l'approximation réelle
- En Python 3, la division donne l'approximation réelle
- Portabilité assurée par l'instruction
  `from __future__ import division`

OPÉRATEURS BINAIRE SUR LES ENTIERS

x | y	Ou binaire
x ^ y	Ou exclusif
x & y	Et binaire
x << n	Décalage à gauche
x >> n	Décalage à droite
~x	Inversion

OPÉRATEURS BINAIRE, EXEMPLE

```
>>> x = 5
>>> y = 6
>>> res = x | y
>>> print res
7
```

5 en binaire 0b101

6 en binaire 0b110

donc 0b101 OU BINAIRE 0b110 donne 0b111 soit 7

PRIORITÉ DES OPÉRATEURS

- Application des règles de priorité mathématique
- Parenthèses ont la plus forte priorité
- À priorité égale, évaluation de gauche à droite
- Opérateurs binaires ont une plus faible priorité que les opérateurs numériques

LES SÉQUENCES

- Types de séquence : string, ~~Unicode~~, list, tuple, (bytearray, buffer et xrange)
- Les chaînes de caractère sont délimités par des simple ou double quotes
- Chaînes de caractères et tuples sont des séquences immuables

LES CHAINES DE CARACTÈRES

- En Python 2, chaines de type str sont encodées en ASCII -> chaines de bytes
- Le type unicode est une chaine encodée en unicode

- En Python 3, chaines de type str sont encodées en unicode
- Portabilité assurée par l'instruction
  from __future__ import unicode_literals

LES SÉQUENCES

```
>>> var_string = 'Une chaîne de caractères'
>>> var_string = "Une chaîne de caractères"
>>> var_string = str("Une chaîne de caractères")

>>> var_unicode = u'Une chaîne de caractères'

>>> var_list = ["Une chaîne", "de", "caractères"]
>>> var_list = list(("Une chaîne", "de", "caractères"))

>>> var_tuple = "Une chaîne", "de", "caractères"
>>> var_tuple = ("Une chaîne", "de", "caractères")
>>> var_tuple = tuple(("Une chaîne", "de", "caractères"))
>>> var_tuple = ("Une chaîne", )
```

OPÉRATIONS SUR LES SÉQUENCES

x in s	True si s contient x, sinon False
x not in s	False si s contient x, sinon True
s + t	Concaténation
s * n	Répétition
s[i]	Élément à l’indice i
s[i, j]	Portion de i à j
s[i, j, k]	Portion de i à j avec le pas k
len(s)	Taille de la chaine
min(s)	Plus petit élément de la séquence
max(s)	Plus grand élément de la séquence
s.index(x)	Indice de la première occurence de x
s.count(x)	Nombre total d’occurrences de x

OPÉRATIONS SPÉCIFIQUES SUR LES CHAINES

str.capitalize()
str.lower()
str.upper()
str.swapcase()
str.expandtabs(size)
str.strip(char)
str.lstrip(char)
str.rstrip(char)
str.join(séquence)
str.isupper()
str.islower()
str.isalnum()
str.isalpha()
str.isdigit()
str.isspace()
str.split(sep)
str.splitlines()

LES VARIABLES : INTERACTION AVEC L'UTILISATEUR (PYTHON 2)

- Saisie de variable avec la fonction input("question ")

- Usage :

```
nom = input('Quel est votre nom ? ')
```

- Input interprète la saisie

- Saisie non interprétée (sous forme de chaîne de caractères) par raw_input("question ")

- Usage :

```
age = int(raw_input("quel est votre âge ? "))
```

LES VARIABLES : INTERACTION AVEC L'UTILISATEUR (PYTHON 3)

- raw_input a été retiré
- input a le comportement du raw_input
- Compatibilité Python 3 dans Python 2 par
  from builtins import input

LES VARIABLES : INTERACTION AVEC L'UTILISATEUR

- Afficher une variable par la fonction print
- Affichage formaté avec l'opérateur % ou la fonction format

AFFICHAGE FORMATÉ AVEC L'OPÉRATEUR %

- Syntaxe :

"Ma variable : %type" % var

"Mes variables : %type, %type" % (var1, var2)

"Résultat : %(val)type %(unit)type" % {'val':var1, 'unit':var2}

- type est d : entier - f : flottant - o : octal - x : hexadécimal - c : caractère - s : chaîne de caractère

AFFICHAGE FORMATÉ : EXEMPLES OPÉRATEUR %

```
>>> reponse = input("Réponse ? ")
Réponse ? 42
>>> pi = 3.14
>>> print(reponse)
42
>>> print("La réponse est %d" % reponse)
La réponse est 42
>>> var = "pi"
>>> print("%s est égal à %f" % (var, pi))
pi est égal à 3.140000
>>> print("%(var)s est égal à %(val)f" % {'var':var, 'val':pi})
pi est égal à 3.140000
```

AFFICHAGE FORMATÉ AVEC LA FONCTION FORMAT

- Syntaxe :
  string.format(*args)

- "Résultat : {}".format(var)

- "Résultat : {}, {}".format(var1, var2)

- "Résultat : {value} {unit}".format(unit=var1, value=var2)

- "Résultat : {:5.2f}".format(var)

- "Résultat : {value:5.2f} {unit}".format(unit=var1, value=var2)

AFFICHAGE FORMATÉ : EXEMPLES FONCTION FORMAT

```
>>> reponse = int(raw_input("Réponse ? "))
Réponse ? 42
>>> pi = 3.14
>>> print("La réponse est {}".format(reponse))
La réponse est 42
>>> var = "pi"
>>> print("{} est égal à {:3.2f}".format(var, pi))
pi est égal à 3.14
>>> nbr = 158.156658
>>> print("nbr ={:06.3e}".format(nbr))
nbr =1.582e+02
```

LES LISTES

- Les listes sont des séquences ordonnées
- Les listes sont des séquences d'objets
- Les listes peuvent être modifiées
- Représentation entre crochets [2, "toto"]

LES LISTES : QUELQUES FONCTION

list.append(objet)	ajoute un objet à la fin de la liste
list.count(valeur)	compte le nombre d'occurrences d'une valeur
list.extend(séquence)	ajoute une séquence à la liste
list.index(valeur, start, stop)	retourne l'index de la première occurrence (start et stop sont optionnel)
list.insert(index, objet)	ajoute un objet avant l'index
list.pop(index)	supprime et retourne l'objet à l'index (index optionnel)
list.remove(value)	supprime la première occurrence
list.reverse()	reverse la liste
list.sort()	trie la liste

SÉQUENCES : ACCÈS À UN ÉLÉMENT

```
>>> sequence = 'Python'
>>> sequence[3]
'h'
>>> sequence[5]
'n'
>>> sequence[-1]
'n'
>>> sequence[-5]
'y'
```

SÉQUENCES : ACCÈS À UN ÉLÉMENT

```
>>> lang = ['Python', 'Java', 'Php', 'Swift']
>>> lang[2]
'Php'
>>> lang[-1]
'Swift'
```

SÉQUENCES : AFFECTATION

```
>>> sequence = 'Python'
>>> lang = ['Python', 'Java', 'Php', 'Swift']
>>> sequence[1] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> lang[1] = 'Scala'
>>> lang
['Python', 'Scala', 'Php', 'Swift']
```

SÉQUENCES : SLICING

- Sémantique : seq[x:y]
- On peut omettre un indice : seq[:y], seq[x:] ou seq[:]
- Si y > len(list) on obtient une liste vide
- On peut utiliser des indices négatifs

SÉQUENCES : SLICING

```
>>> lang = ['Python', 'Java', 'Php', 'Swift']
>>> lang[1:3]
['Java', 'Php']

>>> lang[:3]
['Python', 'Java', 'Php']
>>> lang[-2:]
['Php', 'Swift']

>>> lang[0:10:2]
['Python', 'Php']
```

LES TUPLES

- Les tuples sont des séquences ordonnées
- Les tuples sont des séquences d'objets
- Les tuples ne peuvent pas être modifiées
- Représentation entre parenthèses (2, "toto")
- Attention au tuple singleton : ('toto',)

LES SETS

- Un set est une collection non ordonnée d'éléments non redondants
- On utilise la fonction set() qui prend une séquence en argument
- Supporte les opérateurs ensemblistes

LES SETS

len(s1)	Renvoie la taille de l'ensemble s1
s1.issubset(s2)	Indique si s1 est un sous-ensemble de l'ensemble s2
s1.issuperset(s2)	L'inverse de issubset()
s1.add(ele)	Rajoute un élément à l'ensemble
s1.remove(ele)	Supprime un élément de l'ensemble
s1.pop()	Supprime et renvoie un élément aléatoire de l'ensemble
s1.clear()	Supprime tous les éléments de l'ensemble
s1.union()	Crée un nouveau set issue de l'union de ceux en arguments
s1.intersection()	Fait l'intersection entre les sets

LES DICTIONNAIRES

- Collection non ordonnée de paires clef/valeur
- Dans un dictionnaire chaque clef est unique
- Représenté par des accolades {}
- Les valeurs sont obtenues à partir des clefs
- Pas de notion de position
- Quelques changements en Python 3
  from builtins import dict

LES DICTIONNAIRES : CRÉATION

```
>>> d = dict()
>>> d = {}

>>> d = dict(one=1, two=2)
>>> d = dict([['one', 1], ['two', 2]])

>>> d = {'one': 1, 'two': 2}
```

LES DICTIONNAIRES

dict.clear()	vide le dictionnaire
dict.copy()	créer une copie du dictionnaire
dict.get(clef [, defaut])	retourne la valeur de la clef ou défaut.
dict.has_key(clef)	retourne True si la clef est dans le dictionnaire
dict.items()	retourne une liste de tuple (clef, valeur)
~~dict.keys()~~ ou list(dict)	retourne la liste des clefs
list(dict.values())	retourne la liste des valeurs*
dict.pop(clef)	supprime la paire et retourne la valeur
d[clef] = value	ajoute ou modifie une paire

LES STRUCTURES DE CONTRÔLE

PRINCIPE

- Structurer l'exécution du code
- Bloc exécuté de manière conditionnel ou en boucle
- en tête qui se termine par deux points ":"
- En Python, les blocs sont définis par l'indentation

STRUCTURE CONDITIONNELLE AVEC IF

- if condition: définit l'en-tête conditionnelle
- elif condition: pour une alternative
- else: pour les cas en dehors des conditions précédentes

OPÉRATEURS DE CONTRÔLE

- Opérateurs de comparaison
- Opérateurs logiques

OPÉRATEURS DE COMPARAISON

<	Strictement inférieur à
>	Strictement supérieur à
<=	Inférieur ou égal
>=	Supérieur ou égal
==	Égal à
!=	Différent de

OPÉRATEURS LOGIQUES

Soit X et Y deux expressions

- OU (or)
  si X est vrai, l'expression est vrai sinon l'expression vaut Y

- ET (and)
  si X est faux, l'expression est fausse sinon l'expression vaut Y

- NON (not)
  l'expression est évaluée à l'opposée

TYPES BOOLÉENS

Type	FAUX
Bool	False
int	0
float	0.0
string	""
tuple	()
list	[]
dict	{}
None Type	None

STRUCTURE CONDITIONNELLE

```
>>> x = int(raw_input('Saisissez un entier: '))
Saisissez un entier: 42
>>> if x < 0:
        x = 0
        print 'Negatifs non acceptes'
... elif not x: # équivaut à x == 0
        print('Zero')
... elif x == 42:
        print('La Réponse')
... else:
        print('Merci')

La Réponse
```

EXPRESSION TERNAIRE

- si vrai if condition else si faux

```
>>> statut = "retard" if retard < 10 else "annulé"
```

LES BOUCLES AVEC FOR

- En Python, la boucles for parcourt des séquences
- for element in sequence: définit l'en-tête

LES BOUCLES AVEC FOR

```
>>> cours = ['java', 'python', 'swift']
>>> for c in cours:
        print c, len(c)

java 4
python 6
swift 5
```

LA FONCTION RANGE

- Générateur de liste pour itérer sur séquence d'entiers
- range(x)
- range(x, y)
- range(x, y, z)
- Usage dans les boucles : for i in range(x):

LA FONCTION RANGE : EXEMPLES PYTHON 2

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 2)
[0, 2, 4, 6, 8]
>>> for i in range(3):
        print i

0
1
2
```

LES BOUCLES AVEC WHILE

- En Python, la boucle while s'exécute tant qu'une condition est vrai

- while condition: définit l'en-tête

LES BOUCLES AVEC WHILE

```
>>> a, b = 0, 1
>>> while b < 10:
        print b
        a, b = b, a+b
```

```
1
1
2
3
5
8
```

BREAK, CONTINUE ET ELSE

- break : interrompt l'exécution d'une boucle et la quitte
- continue : interrompt l'exécution d'une boucle et passe à l'itération suivante
- else : le bloc s'exécute après la boucle sauf si interrompue par un break.
- pass : ne fait rien, utilisé quand une instruction est nécessaire

COMPREHENSION LISTS

- Listes en intension
- https://fr.wikipedia.org/wiki/Intension_et_extension

- En logique, l'intension d'un concept est sa définition
- Les listes en intension permettent de définir la transformation d'une liste

COMPREHENSION LISTS

```
>>> sequence = ["a", "b", "c"]
>>> new_sequence = []
>>> for element in sequence:
        new_sequence.append(element.upper())
```

COMPREHENSION LISTS

```
>>> sequence = ["a", "b", "c"]
>>> new_sequence = [element.upper() for element in sequence]
```

COMPREHENSION LISTS

- Structure générale
  [transformation for élément in collection if condition]

- La condition est optionnelle, si présente, la liste est filtrée en fonction de la condition

- La transformation peut retourner n'importe quel type

COMPREHENSION LISTS

```
>>> sommes = []
>>> for nombre in range(10):
        if nombre % 2 == 0:
            sommes.append(sum(range(nombre)))
```

```
>>> print(sommes)
[0, 1, 6, 15, 28]
```

COMPREHENSION LISTS

```
>>> print([sum(range(nombre)) for nombre in range(10) if nombre % 2 == 0])
[0, 1, 6, 15, 28]
```

PYTHON : MODULES ET PACKAGES

PYTHON : MODULES ET PACKAGES

- Persister le code
- Organiser son code
- Module : fichier
- Package : arborescence de répertoires

LES MODULES

- Fichier texte
- Fichier avec extension .py
- Doit contenir en en-tête le shebang et l'encodage

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
```

LES MODULES

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

print("Hello World")
```

LES PACKAGES

- Répertoire destiné à contenir un autre package et/ou un ou plusieurs modules
- Doit contenir un fichier __init__.py
- Le nom du répertoire est le nom du package

IMPORTER LE CONTENU D'UN PACKAGE DANS UN MODULE

- Mot clef : import

- Importer référence à un package avec
  import mon.package

- Importer tout le contenu du package avec from et *
  from mon.package **import** *

- Importer des éléments spécifiques du package ou module
  from mon.package **import** monModule, mon AutreModule
  from mon.package.monModule **import** MaClasse

IMPORTER LE CONTENU D'UN PACKAGE DANS UN MODULE

```
In [1]: import math
In [2]: print(math.pi)
3.141592653359

In [3]: from math import pi, sqrt
In [4]: print(pi, sqrt)
3.141592653359 <built-in function sqrt>

In [5]: from math import *
In [6]: print(pi, sqrt)
3.141592653359 <built-in function sqrt>
```

LE MODULE __MAIN__

- Une variable spéciale des modules : __name__
- __name__ contient le nom du module
- SAUF pour le module principal, __name__ == '__main__'
- Lorsqu'on importe un module, Python interprète le module importé
- Pour n'exécuter du code que dans le module principal : if __name__ == '__main__':

LE MODULE __MAIN__

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def whoYaGonnaCall():
    print("Ghost Busters !")

whoYaGonnaCall()
```

LE MODULE __MAIN__

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def whoYaGonnaCall():
    print("Ghost Busters !")

if __name__ == '__main__':
    whoYaGonnaCall()
```

LES PACKAGES, FICHIER __INIT__.PY

- Fichier qui doit être vide
- __init__.py est interprété lors de l'import du package
- Contenu du __init__.py doit configurer le package
- Variable spéciale __all__ déclare la liste des modules importés par *

```
__all__ = ['mon_module', 'mon_autre_module']
```

STRUCTURE D'UN PROJET

- http://docs.python-guide.org/en/latest/writing/structure/
- https://github.com/kennethreitz/samplemod
- Recommendation de Kenneth Reitz

DISTRIBUTION DES PACKAGES

- Python fournit un outil pour créer un installeur
- Par convention, module setup.py
- Permet création d'archive par
  python setup.py sdist
- Permet une installation par
  python setup.py install

DISTRIBUTION DES PACKAGES

```
from distutils.core import setup
setup (
    name='Nom du Package',
    version='1.0',
    description='Package pour faire ...',
    author='Nom Prenom',
    author_email='name@example.com',
    packages=['Nom du Package'],
)
```

DISTRIBUTION DES PACKAGES

Il n'est pas nécessaire d'installer un package pour y accéder

- Placer dans un chemin et adapter le PYTHONPATH
- Placer le package dans le répertoire du projet python

PYTHON : LES BASES

SHELL INTERACTIF ET MODULES

SHELL INTERACTIF ET MODULES

- On accède au contenu d'un module avec l'instruction import
- Si on modifie le module, la référence importée n'est pas mise à jour
- On met à jour un module avec la fonction reload
- Les instances ne sont pas mises à jour
- Python 3 : importlib.reload()

SHELL INTERACTIF ET MODULES

```
>>> from formation import exolifo as lifo
>>> lifo.pile('Galaad', 'Robin', 'Bedevere')
>>> lifo.depile()
... 'Bedevere'
>>> reload(lifo)
```

PYTHON :
LES ARGUMENTS DE LIGNE DE COMMANDE

COMMENT EXÉCUTER UN PROGRAMME PYTHON

- Exécuter Python en passant en paramètre le module à exécuter

  > python mon_module.py

- Si le module est exécutable et possède un shebang, en exécutant le module

  > mon_module.py

- Il est possible de passer des arguments en ligne de commande

  > mon_module.py -v -o output.txt

RÉCUPÉRER LES ARGUMENTS DE LA LIGNE DE COMMANDE

```
import sys

if __name__ == '__main__':
    for arg in sys.argv:
        print(arg)
```

LES ARGUMENTS DE LA LIGNE DE COMMANDE AVEC GETOPT

```
import getopt, sys
try:
    opts, args = getopt.getopt(sys.argv[1:], 'ho:', ['help', 'output='])
except getopt.GetoptError as err:
    sys.exit(2)
for o, a in opts:
    if o in ('-h', '--help'):
        usage()
        sys.exit()
    elif o in ('-o', '--output'):
        output_file = a
```

LES FONCTIONS

DÉCLARATION

- mot clef def
- nom de la fonction
- liste des paramètres entre parenthèse
- En-tête terminé par deux points ":"
- bloc d'instruction indenté
- mot clef return permet de retourner un résultat
- Si return est omis ou n'a pas de valeur de retour : None

EXEMPLE

```
>>> def fib(n):
        a, b = 0, 1
        while a < n:
            print(a)
            a, b = b, a + b

>>> fib(10)
0
1
1
2
3
5
8
```

PARAMÈTRES ET RETOUR

- Paramètre et retour ne sont pas typés
- Une fonction peut retourner plusieurs éléments

PARAMÈTRES OPTIONNELS ET NOMMÉS

- Un paramètre peut être optionnel
- Une valeur par défaut doit lui être attribué dans la signature
- La valeur peut être affectée par le nom du paramètre

PARAMÈTRES ET RETOUR

```
>>> def fib(n, floor=0):
        a, b = floor, floor + 1
        result = []
        while a < n:
            result.append(a)
            a, b = b, a + b
        return floor, result

>>> fib(10)
(0, [0, 1, 1, 2, 3, 5, 8])
>>> fib(20, 5)
(5, [5, 6, 11, 17])
>>> fib(20, floor=4)
(4, [4, 5, 9, 14])
```

NOMBRE D'ÉLÉMENTS ARBITRAIRES

- Passage d'un tuple, le nom est précédé d'une étoile *args
- Passage d'un dictionnaire, le nom est précédé de deux étoiles **kargs

NOMBRE D'ÉLÉMENTS ARBITRAIRES

```
>>> def multi_args(*args, **kargs):
        for n in args:
            print n
        for k in kargs.keys():
            print k, ":",   kargs[k]
```

PORTÉE DES VARIABLES

- Les variables ont une portée dans leur espace local
- Une variable déclarée dans une fonction n'est visible qu'à l'intérieur de la fonction
- Une variable déclarée en dehors de la fonction (variable globale) est visible dans la fonction si elle a été déclarée avant en lecture seule
- Une variable globale peut être modifiée dans une fonction si elle est déclarée par le mot-clef global

PORTÉE DES VARIABLES

```
>>> var = 10
>>> def func():
        global var
        var = 11
>>> var
10
>>> func()
>>> var
11
```

LES FONCTIONS EN PARAMÈTRE ET VARIABLE

- Une fonction est un objet…
- ma_fonction() interprète la fonction
- ma_fonction est une référence à la fonction

LES FONCTIONS EN PARAMÈTRES

```
>>> def tab(fonction, inf, sup, pas):
        for i in range(inf , sup, pas):
            y = fonction(i)
            print("f({}) = {}".format(i, y))
```

```
>>> def maFon(x) :
        return 2*x + 3

>>> tab(maFon, -4, 4, 2)
f(-4) = -5
f(-2) = -1
f(0) = 3
f(2) = 7
```

FONCTIONS ANONYMES (LAMBDAS)

- fonctions jetables
- définir et utiliser une fonction anonyme d’une traite
- Définition par le mot clef lambda
- Ne peuvent être écrites que sur une ligne
- Ne peuvent contenir qu'une seule instruction

FONCTIONS ANONYMES (LAMBDAS)

```
def carre(val):
    return val * val
```

```
carre = lambda val: val * val
```

FONCTIONS ANONYMES (LAMBDAS)

```
>>> def tab(fonction, inf, sup, pas):
        for i in range(inf , sup, pas):
            y = fonction(i)
            print("f({}) = {}".format(i, y))
```

```
>>> tab(lambda x: 2*x + 3, -4, 4, 2)
f(-4) = -5
f(-2) = -1
f(0) = 3
f(2) = 7
```

LES EXPRESSIONS GÉNÉRATRICES

```
>>> nombres = [sum(range(nombre)) for nombre in range(0, 10, 2)]
>>> for nombre in nombres:
        print nombre
```

```
>>> nombres = [sum(range(nombre)) for nombre in range(0, 10, 2)]
>>> type(nombres)
```

LES EXPRESSIONS GÉNÉRATRICES

```
>>> nombres = (sum(range(nombre)) for nombre in range(0, 10, 2))
>>> for nombre in nombres:
        print nombre
```

```
>>> nombres = (sum(range(nombre)) for nombre in range(0, 10, 2))
>>> type(nombres)
```

LES EXPRESSIONS GÉNÉRATRICES

- Un générateur est un iterable
- Un générateur ne contient pas de valeurs
- Un générateur calcule chaque valeur à la volée
- Un générateur ne peut être parcouru qu'une seule fois

LES EXPRESSIONS GÉNÉRATRICES

- Dans une fonction, le mot-clef yield transforme la fonction en générateur
- yield est utilisé à la place de return
- L'appel à une fonction génératrice n'exécute pas la fonction mais retourne un objet générateur

LES EXPRESSIONS GÉNÉRATRICES

```
>>> import sys
>>> def gen_fibonacci (max = sys.maxsize) :
        a, b = 0, 1 while a < max:
            a, b = b, a+b
            yield a
```

```
>>> for n in gen_fibonacci(1000):
        print n
```

PROGRAMMATION ORIENTÉE OBJET

LES PARADIGMES DE PROGRAMMATION

Il s'agit des différentes façons de raisonner et d'implémenter une solution à un problème en programmation.

- La programmation impérative
  paradigme originel et le plus courant
- La programmation orientée objet (POO)
  consistant en la définition et l'assemblage de briques logicielles appelées objets
- La programmation déclarative
  consistant à déclarer les données du problème, puis à demander au programme de le résoudre

LA PROGRAMMATION ORIENTÉE OBJET (POO)

- Repose sur la définition et l'assemblage de briques logicielles appelées objets
- Le problème à résoudre est modélisé par les objets
- Chaque objet a une et une seule responsabilité

LES OBJETS

Caractérisés par

- Un état
- Des comportements

LES CLASSES

Sont les définition des objets

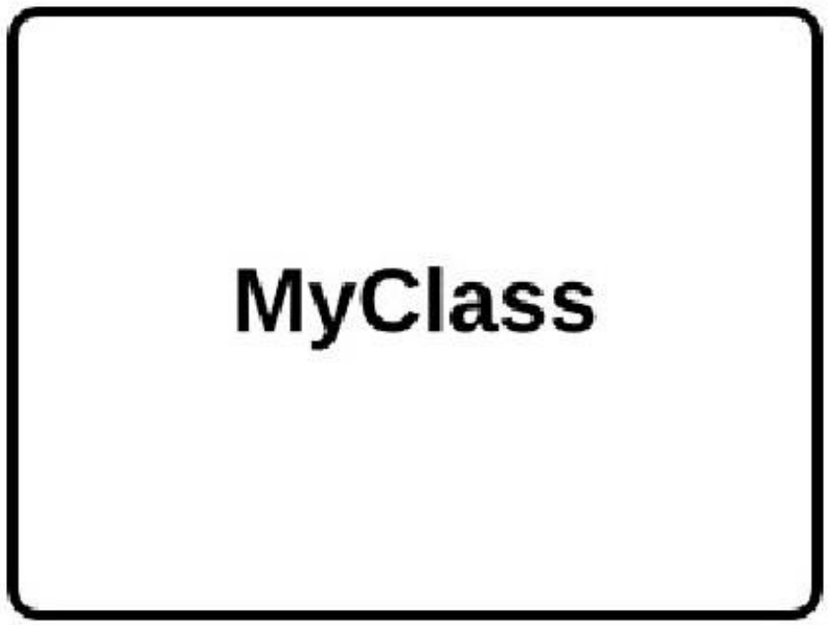
- Un objet est une instance d'une classe
- En POO, nous définissons des classes
- En POO, nous manipulons des instances des classes
- Le type d'un objet est sa classe

MODÉLISATION ET REPRÉSENTATION

UML : Unified Modeling Language

- Langage de modélisation graphique
- Basé sur des pictogrammes
- Standardisé
- En 2.3 propose 14 types de diagrammes

REPRÉSENTER UNE CLASSE



DÉCLARATION

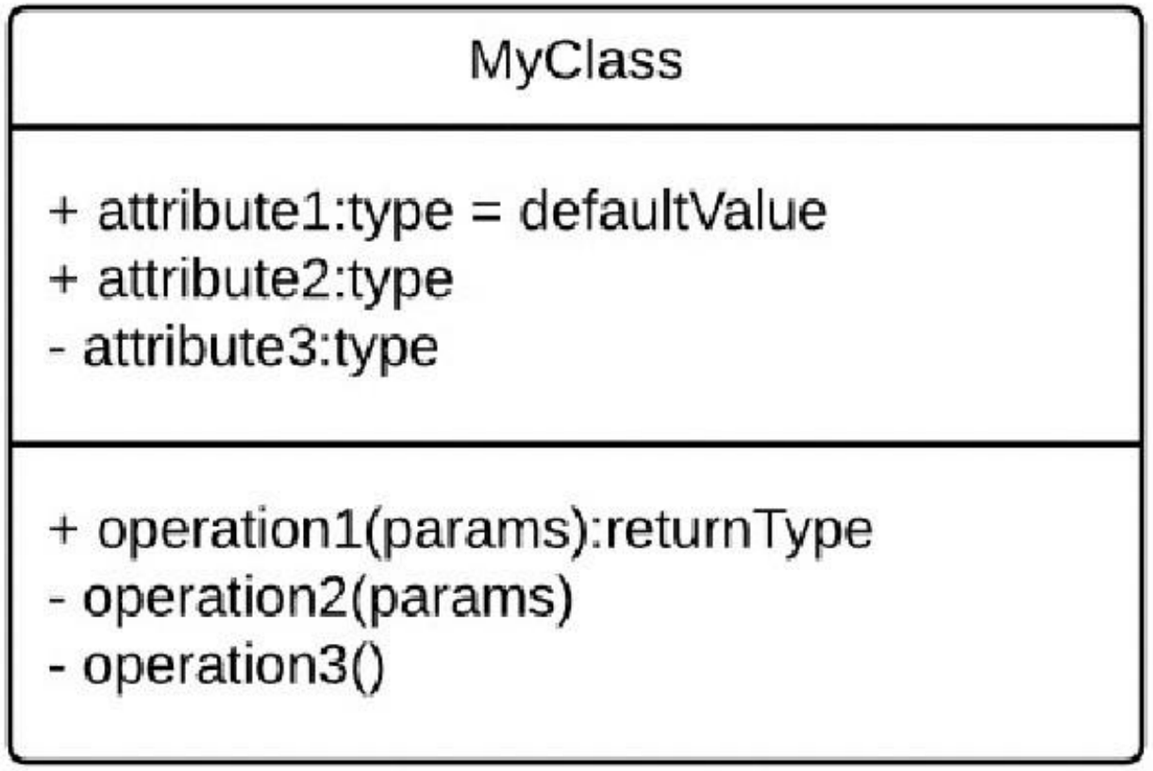
- mot clef class
- nom de la classe (CamelCase)
- En-tête terminé par deux points ":"
- contenu de la classe indenté
- instanciation par le nom de classe suivi de parenthèses

DÉCLARATION

```
>>> class MaClasse:
        pass

>>> maClasse = MaClasse()
```

REPRÉSENTER UNE CLASSE



LES ATTRIBUTS

- Attributs de classe
  - Appartiennent à la classe
  - Accès en préfixant avec le nom de la classe
- Attributs d'instance
  - Appartiennent à l'instance
  - Accès en préfixant par la référence de l'objet self

ATTRIBUTS DE CLASSES

```
>>> class Test:
        val=0
        def increment(self):
            Test.val += 1

>>> t1 = Test()
>>> t2 = Test()
>>> t1.val
0
>>> t2.val
0
>>> t1. increment()
>>> t1.val
1
>>> t2.val
1
```

LES MÉTHODES

- fonctions définies dans les classes
- premier paramètre est toujours self
- self est une référence d'instance

LES MÉTHODES

```
>>> class MaClasse:
        def ma_methode(self):
            print 'hello world'

>>> maClasse = MaClasse()
>>> maClasse.ma_methode()
hello world
```

INSTANCIER UN OBJET

- Se fait par l'appel d'un constructeur
- Python appelle successivement un constructeur (__new__) et un initialiseur (__init__)
- En Python, on surcharge l'initialiseur __init__
- Existe un destructeur __del__ appelé avant la destruction de l'instance

INSTANCIER UN OBJET

```
>>> class UselessClass:
        def __init__(self):
            pass

>>> class Employee:
    def __init__(self, first_name, last_name):
        self._first_name = first_name
        self._last_name = last_name
```

VISIBILITÉ DES ATTRIBUTS ET MÉTHODES

- En Python, tout a une visibilité publique
- Convention : si le nom d'un attribut ou d'une méthode commence par un underscore, c'est un élément privé

- def ma_methode(self) définit une méthode publique
- def _ma_methode(self) définit une méthode privée

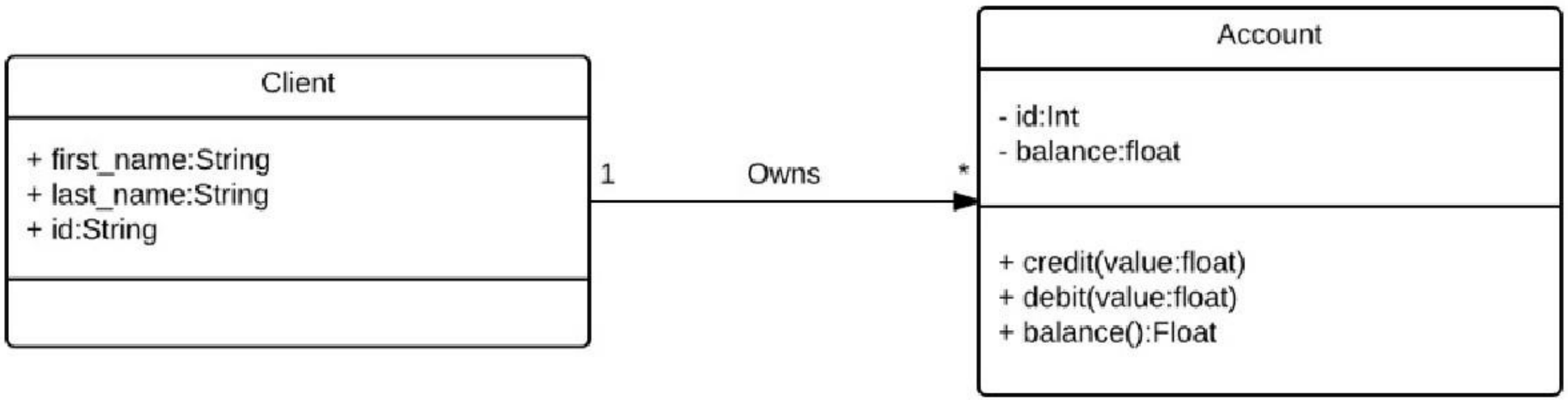
MÉTHODES SPÉCIALES

__init__	Initialiseur appelé juste après l'instanciation de l'objet
__del__	Appelé juste avant la destruction de l'objet
__str__	Appelé par la fonction de conversion de type str() et la fonction print
__lt__	x < y
__le__	x <= y
__eq__	x = y
__ne__	x != y
__ge__	x >= y
__gt__	x > y

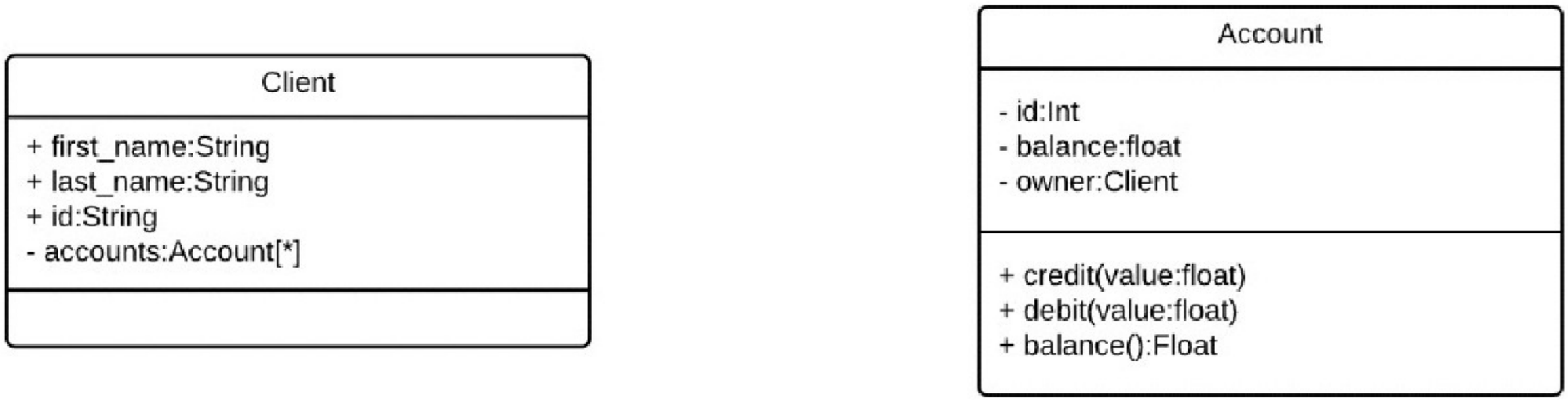
MÉTHODES SPÉCIALES

__neg__	-x
__add__	x + y
__sub__	x - y
__mul__	x * y
__div__	x / y

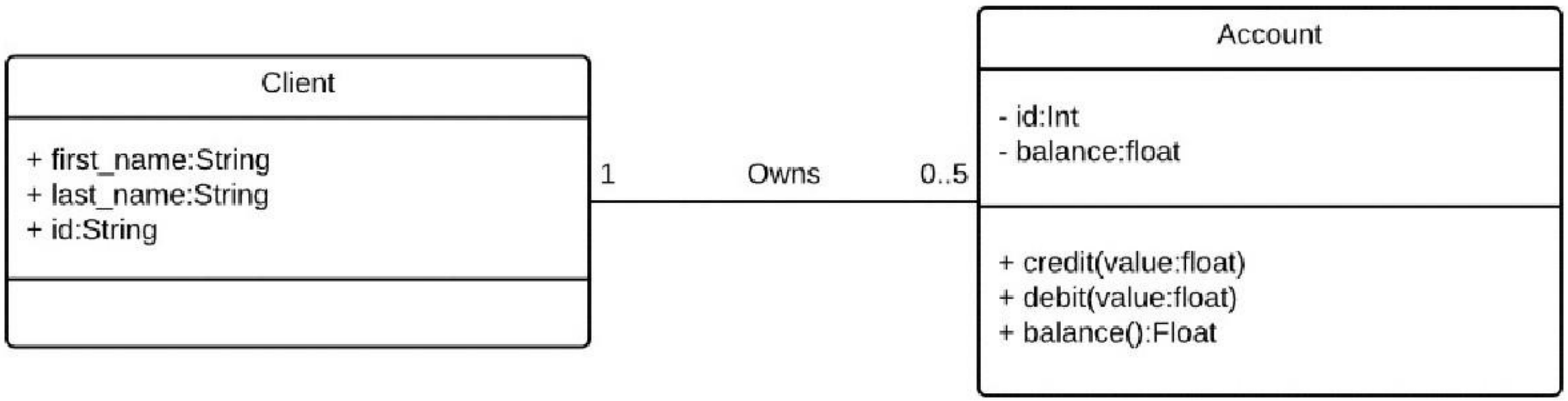
RELATIONS ENTRE CLASSES : ASSOCIATION



RELATIONS ENTRE CLASSES : ASSOCIATION



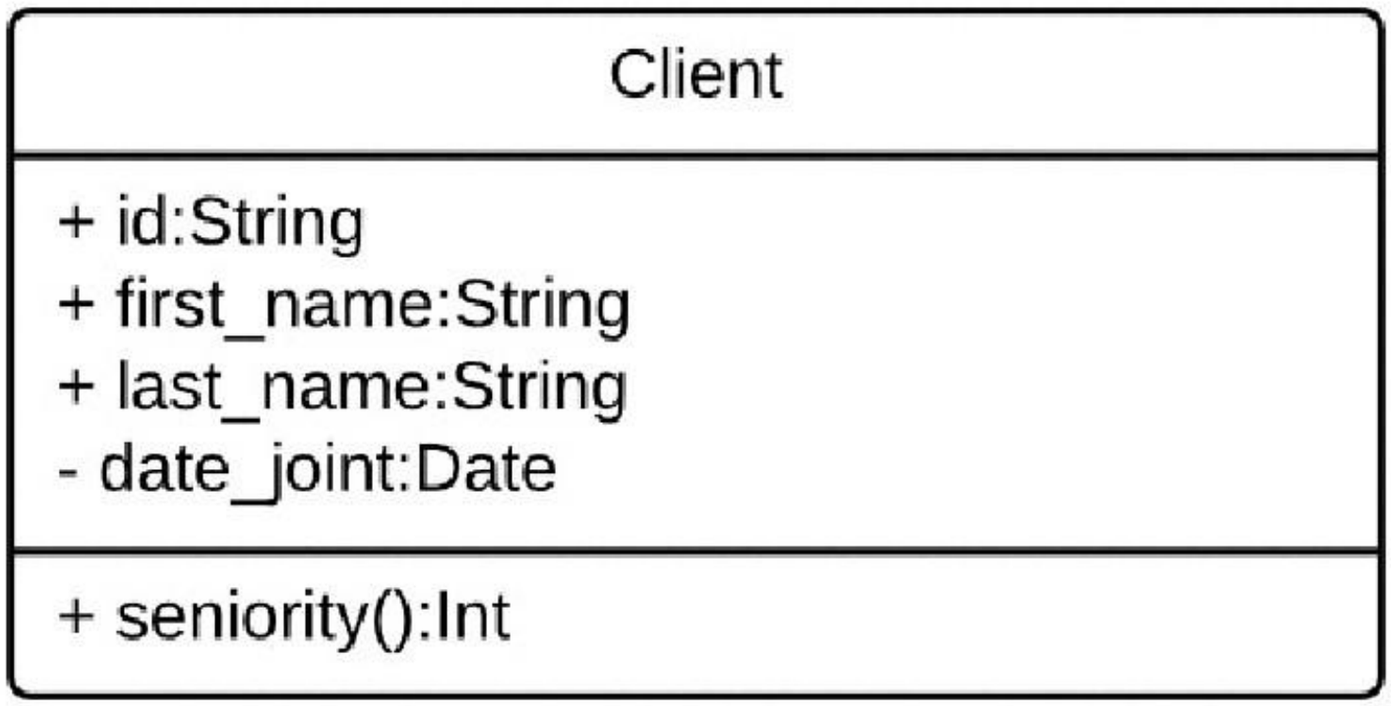
RELATIONS ENTRE CLASSES : ASSOCIATION



PRINCIPE D'ENCAPSULATION

- Données et traitement sont réunies sous la même entité
- L'implémentation est masquée de l'extérieur
- On communique avec l'objet par les attributs et méthodes publiques
- Permet le minimum de modification du code existant lors de l'évolution
- Une classe est ouverte à l'extension et fermée à la modification

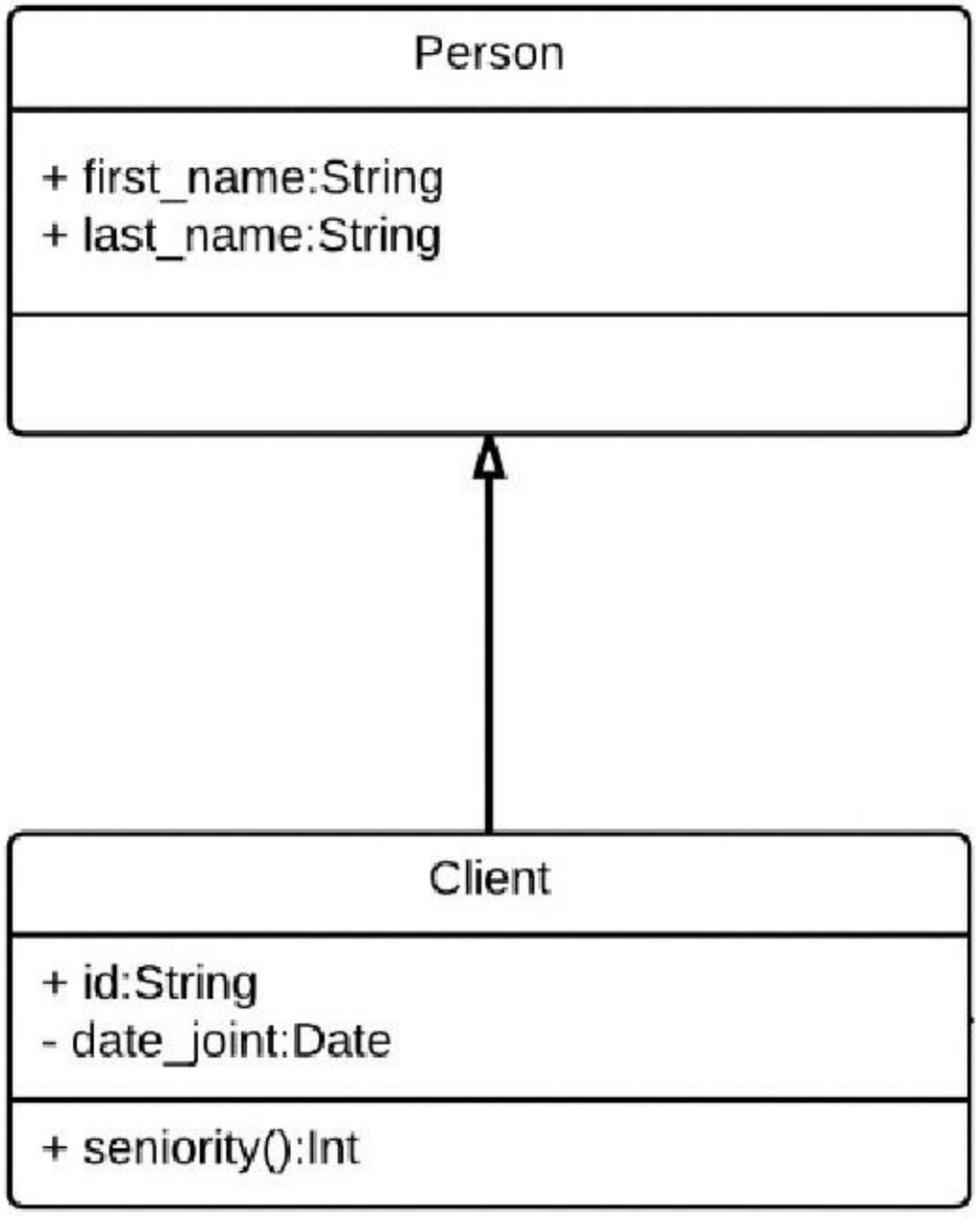
PRINCIPE D'ENCAPSULATION



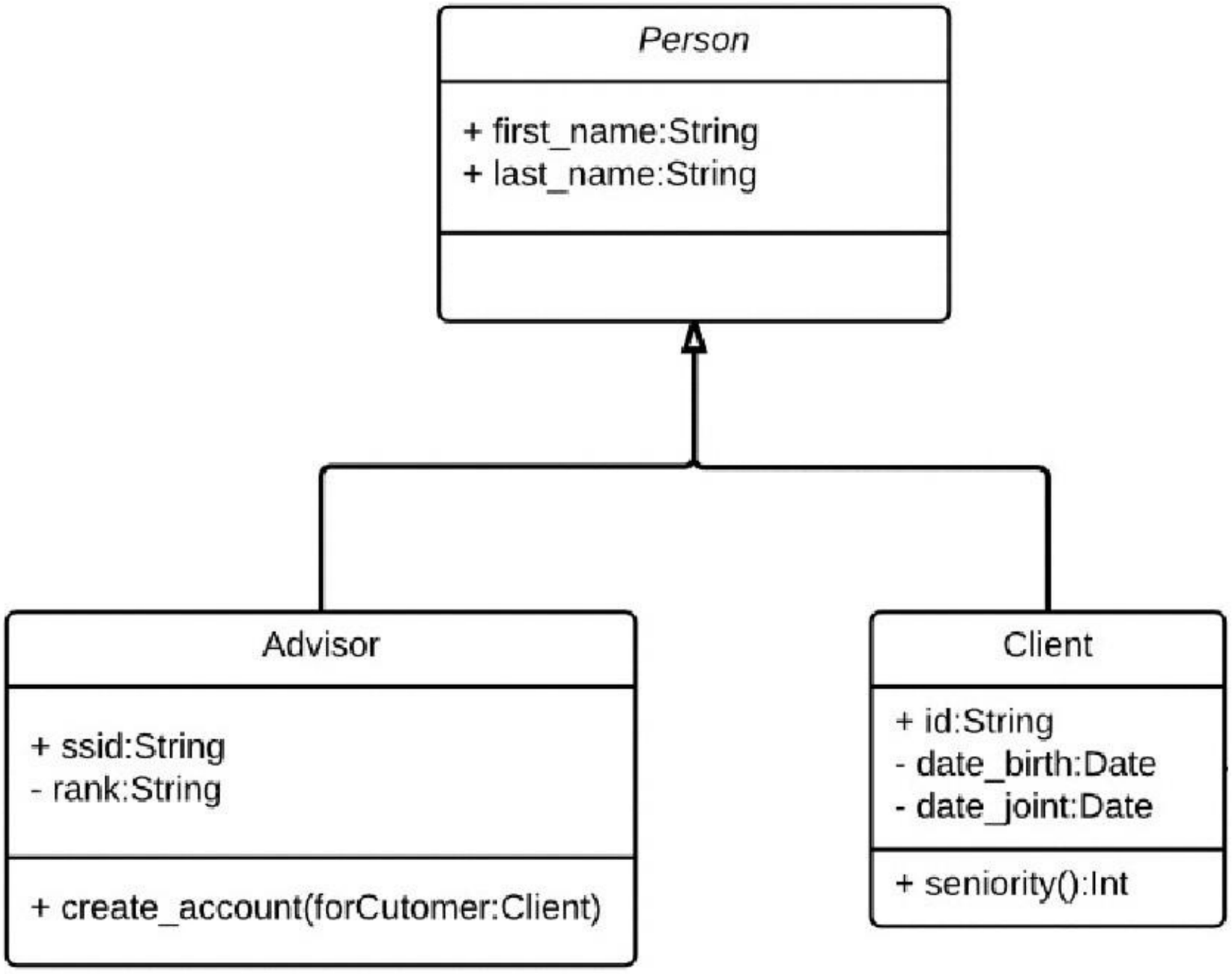
L'HÉRITAGE

- Propriété de généraliser ou spécialiser des états ou comportements
- Généralisation : définition unique, évite duplication
- Spécialisation : adapter caractéristiques et comportements
- Abstraction
- Polymorphisme

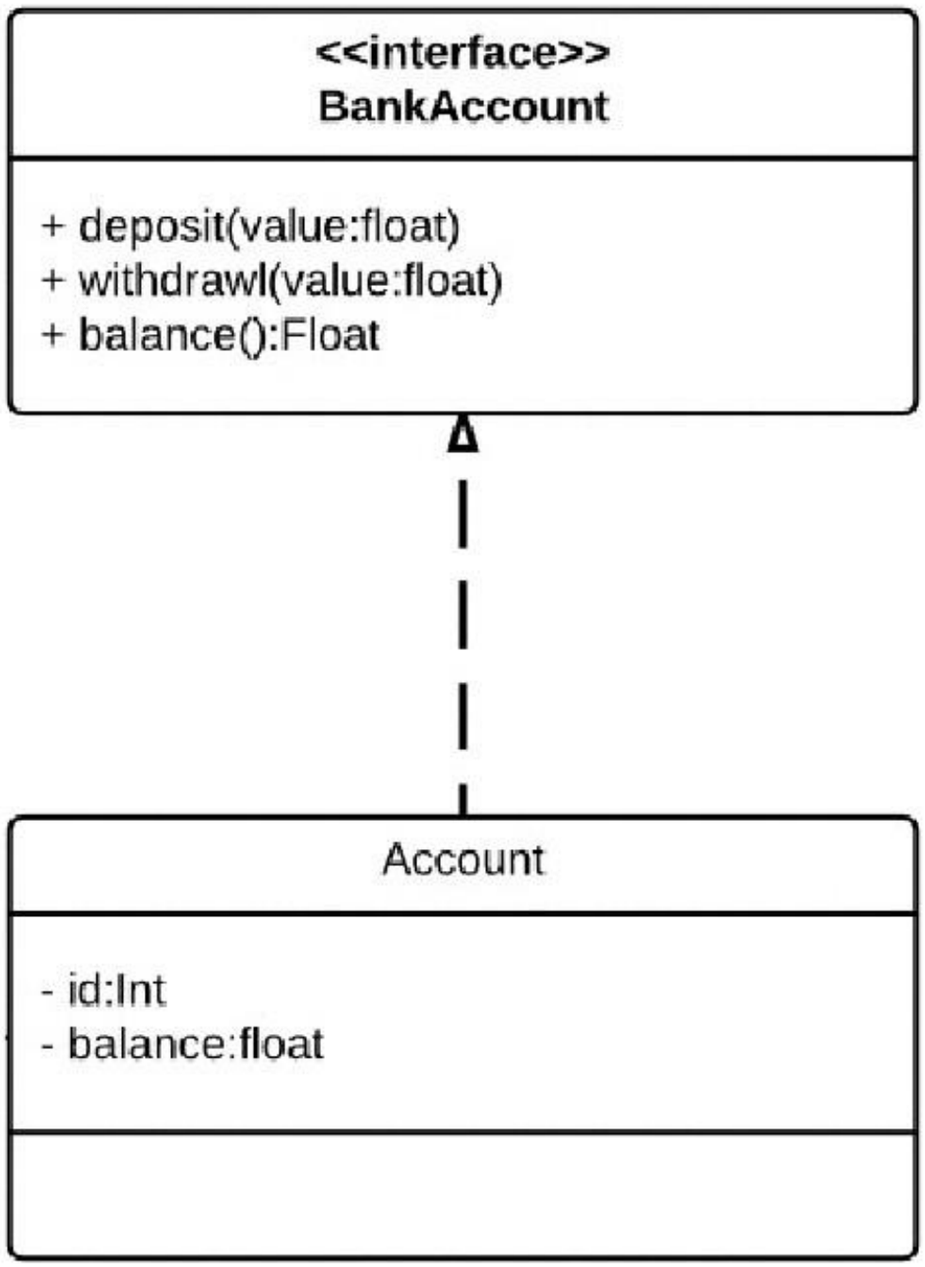
L'HÉRITAGE


Person
+ first_name:String
+ last_name:String
Client
+ id:String
- date_joint:Date
+ seniority():Int

L'HÉRITAGE


Person
+ first_name:String
+ last_name:String
Advisor
+ ssid:String
- rank:String
+ create_account(forCutomer:Client)
Client
+ id:String
- date_birth:Date
- date_joint:Date
+ seniority():Int

ABSTRACTION : INTERFACES



L'HÉRITAGE

- Déclaration de la classe suivi du mon de la classe héritée entre parenthèses
  class Client(Personne):
- Python supporte l'héritage multiple
  class Vendeur(Personne, Employe):
- Toutes les classes héritent de object
- class MaClasse: est équivalent à class MaClasse(object):

L'HÉRITAGE

- Appel des méthodes ou attributs du parent doit être préfixé par le nom de la classe parente

```
class Client(Personne):
    def __init__(self, nom, prenom, n_fidelite):
        Personne.__init__(self, nom, prenom)
        self._n_fidelite = n_fidelite
```

POLYMORPHISME

- Capacité à redéfinir un comportement
- Capacité du système à choisir dynamiquement la méthode qui correspond au type réel de l'objet en cours

- Exemple :
  - Compte à débit immédiat, débit() débite le solde
  - Compte à débit différé, débit() ne débite pas le solde

POLYMORPHISME ET DUCK TYPING

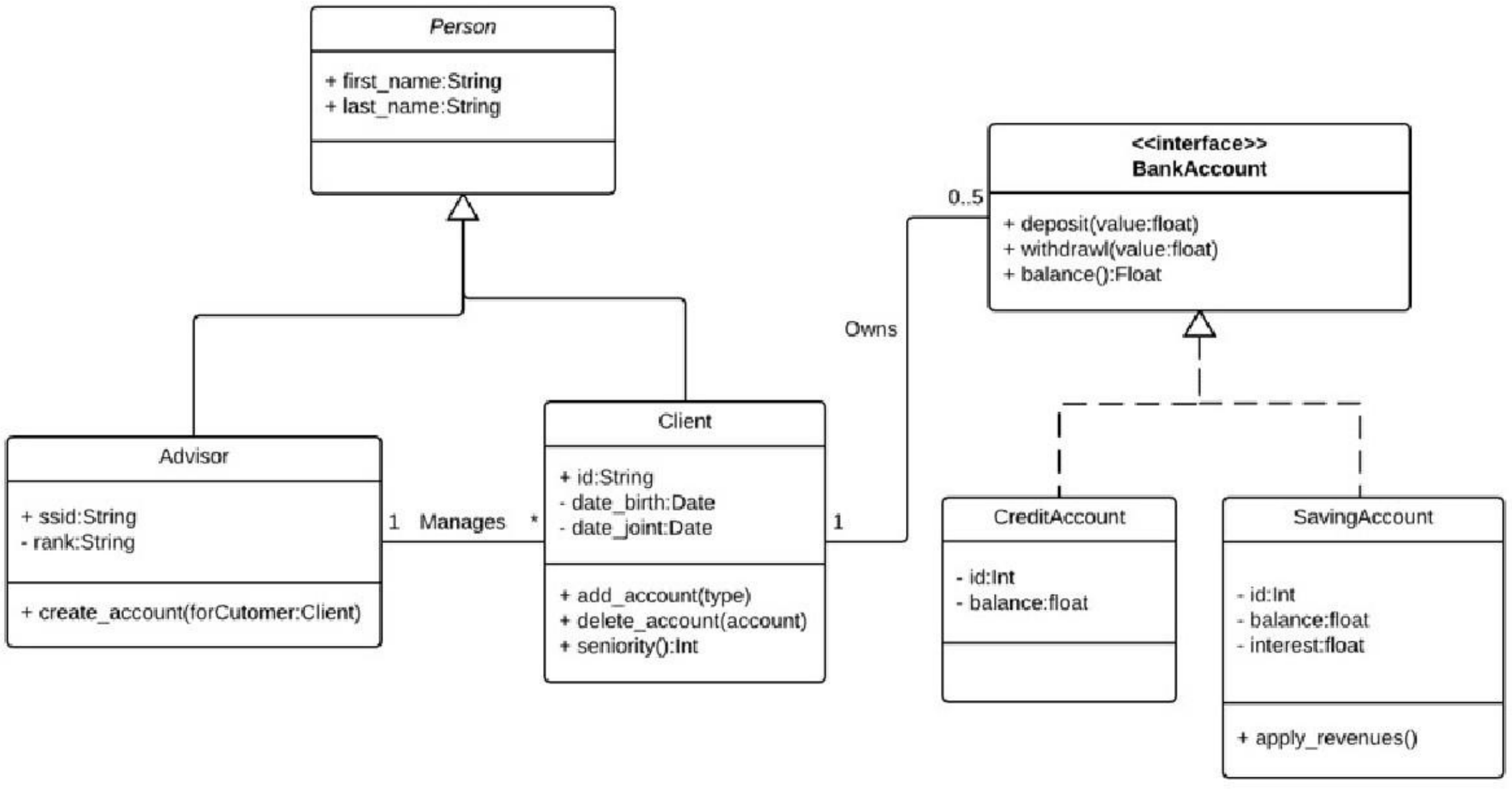
- Capacité à spécialiser un comportement
- En Python, le polymorphisme repose sur le Duck Typing.
- "Si je vois un animal qui vole comme un canard, cancane comme un canard, et nage comme un canard, alors j'appelle cet oiseau un canard"

POLYMORPHISME ET DUCK TYPING

```
>>> def triple_and_one(a, b):
        print a * 3 + b

>>> triple_and_one(5, 2)
17
>>> triple_and_one("hip ", "houra")
hip hip hip houra
```

EXEMPLE COMPLET



CAS D'UTILISATION

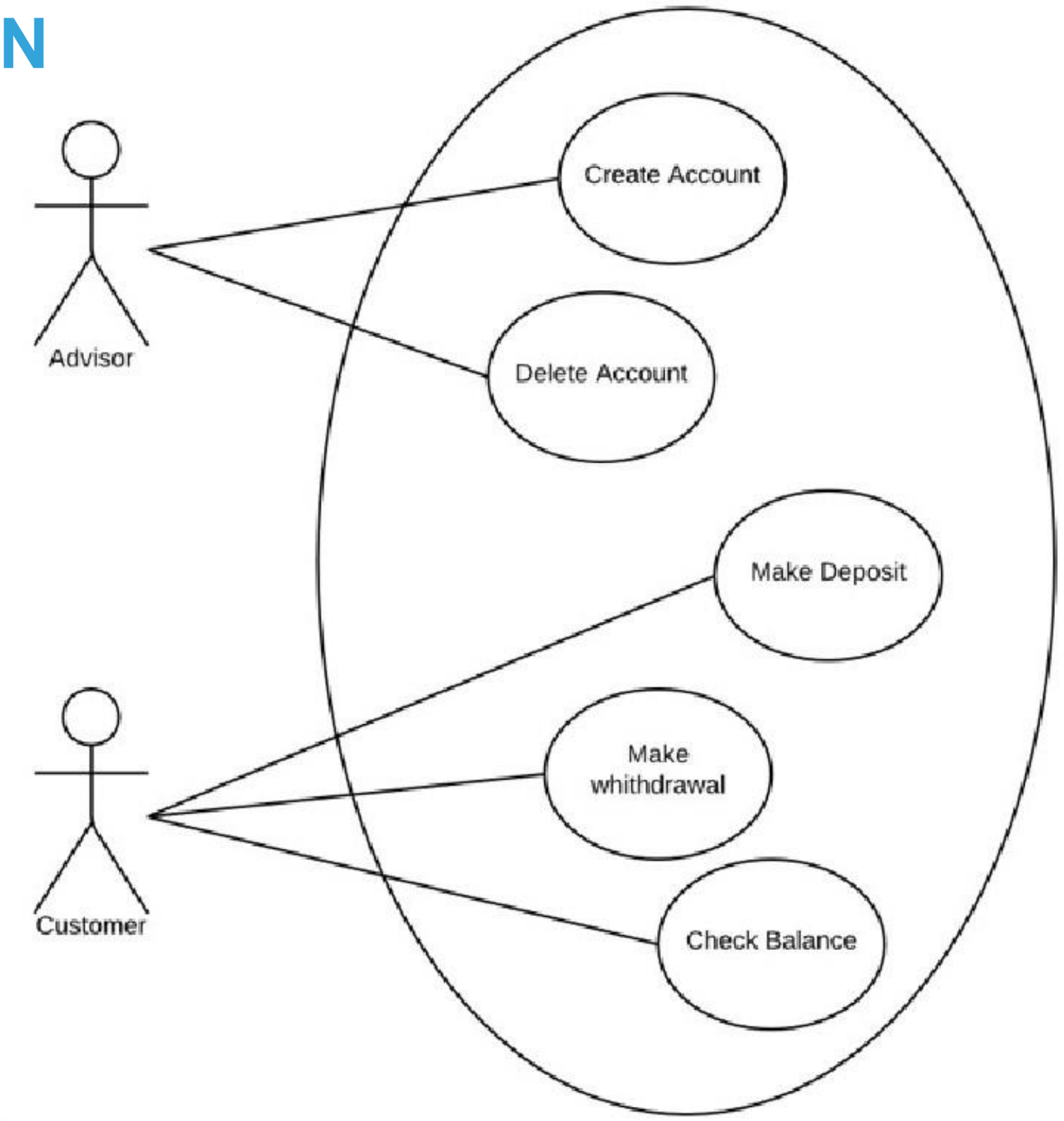
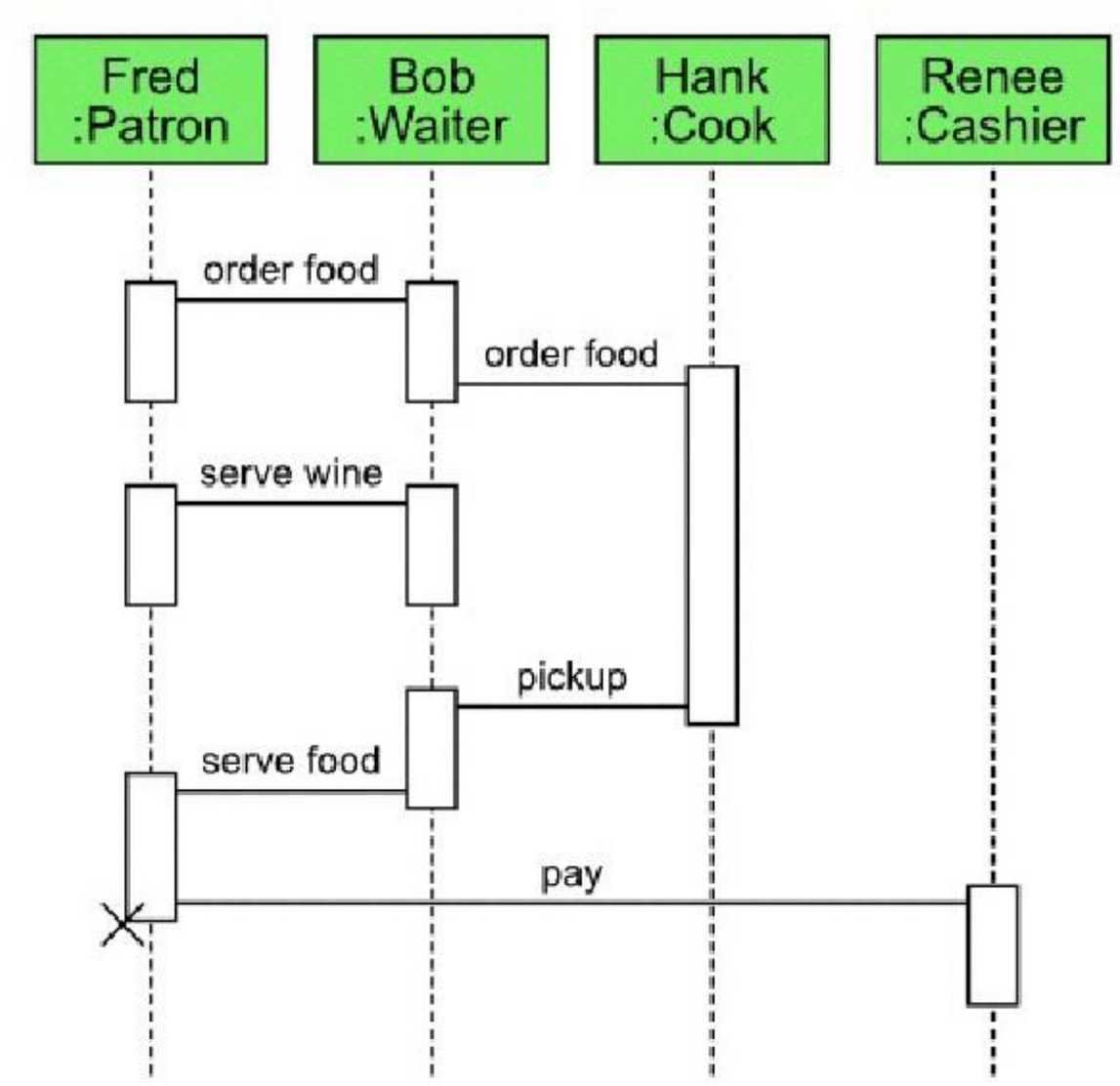


DIAGRAMME SÉQUENCE



https://fr.wikipedia.org/wiki/Diagramme_de_s%C3%A9quence

L'INTROSPECTION

- Rappel, les fonctions et méthodes help(), dir(), __dict__, __class__, __class__.__name__, __class__.__bases__, getattr(objet, méthode, valeur par défaut)…
- Pour comparer des objets, ne pas oublier isinstance(objet)

PYTHON :

LES EXCEPTIONS

LES EXCEPTIONS

- Mécanisme d'interruption du programme pour signaler que quelque chose d'anormal se produit.
- Mécanisme qui délègue au bloc appelant la gestion de l'exception.
- "It’s easier to ask for forgiveness than permission"

PRINCIPE

- Lorsqu'une exception se produit, elle stoppe l'exécution du programme et retourne une Exception.
- Une exception non gérée interrompt le programme qui affiche la stack trace.

EXCEPTIONS NON CAPTURÉES

```
>>> import nawak
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named nawak
```

```
>>> d = {'cle': 'valeur'}
>>> d['nope']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'nope'
```

CAPTURE DES EXCEPTIONS

- Délimiter le code pouvant lever une exception par les instructions try et except
- Si une exception est levée, le bloc except est appelé
- Le bloc except doit déclarer les exceptions gérées

CAPTURE DES EXCEPTIONS

```
# i est définit plus haut
personnages = ['Luke', 'Han', 'Yoda', 'Leia']
try:
    resultat = personnages[i]
# i est plus grand que la taille du tableau
except IndexError:
    resultat = None
```

CAPTURE DES EXCEPTIONS

Plusieurs exceptions peuvent être gérées d'un bloc

- En les cumulant
- Avec plusieurs clauses except

CAPTURE DES EXCEPTIONS

```
# i est définit plus haut
personnages = ['Luke', 'Han', 'Yoda', 'Leia']
try:
    resultat = personnages[100/i]
except (IndexError, ZeroDivisionError):
    resultat = None

except TypeError:
    print("The Force is weak in this one")
```

CAPTURE DES EXCEPTIONS

- Par principe, capturez toujours les exceptions au plus proche de la logique du programme.
- En omettant un nom d'exception après la clause except, on attrape toutes les exceptions : mauvaise pratique

ELSE ET FINALLY

- else permet d'exécuter du code uniquement lorsqu'aucune exception n'est levée

- finally est exécuté après tous les autres blocs qu'il y ai eu exception ou non

- Le bloc finally sert généralement à faire du nettoyage.

DÉCLENCHER UNE EXCEPTION

- Instruction raise
  raise NameError

- L'exception lancée doit hériter de l'objet Exception

- raise permet de relancer une exception

- Une exception peut contenir des arguments

DÉCLENCHER UNE EXCEPTION

```
def votre_super_fonction(param):
    if param not in (1, 2, 3):
        raise ValueError("'param' can only be either 1, 2 or 3")
    # reste du code
```

```
>>> votre_super_fonction(4)
Traceback (most recent call last):
  File "<ipython-input-3-bcd6e8653c83>", line 1, in <module>
    votre_super_fonction(4)
  File "<ipython-input-2-46fc7cd18c42>", line 3, in
votre_super_fonction
    raise ValueError("'param' can only be either 1, 2 or 3")
ValueError: 'param' can only be either 1, 2 or 3
```

DÉFINIR SES PROPRES EXCEPTIONS

```
class MonErreur(Exception):
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return repr(self.value)
```

LE MOT CLEF WITH

- Proposé par les Context Manager
- Décorateur

- Un Context Manager gère des actions avant et/ou après l'appel du bloc with.

LE MOT CLEF WITH

```
try:
    fichier = open('/tmp/fichier', 'w')
except (IOError, OSError):
    # gérer l'erreur
else:
    # faire un truc avec le fichier
finally:
    try:
        fichier.close()
    except NameError:
        pass
```

LE MOT CLEF WITH

```
try:
    with open('/tmp/fichier', 'w') as fichier:
        # faire un truc avec le fichier
except (IOError, OSError):
    # gérer l'erreur
```

CRÉER SON CONTEXT MANAGER

```
class MonSuperContextManager(object):
     def __enter__(self):
          print "Avant"
     def __exit__(self, type, value, traceback):
          # faites pas attention aux paramètres, ce sont
          # toutes les infos automatiquement passées à
          # __exit__ et qui servent pour inspecter
          # une éventuelle exception
          print "Après"

with MonSuperContextManager():
     truc()
```

PYTHON : QUALITÉ

QU'EST CE QUE LA QUALITÉ ?

- Conformité aux exigences et aux attentes établies
- Ensemble des actions permettant d'assurer la fiabilité, la maintenance et l'évolutivité du logiciel
- Suivie par l'ensemble des mesures mises en place

POURQUOI LA QUALITÉ ?

- les délais de livraison des logiciels sont rarement tenus, le dépassement de délai et de coût moyen est compris entre 50 et 70 %

- la qualité du logiciel correspond rarement aux attentes, le logiciel ne correspond pas aux besoins, il consomme plus de moyens informatiques que prévu, et tombe en panne

- les modifications effectuées après la livraison d'un logiciel coûtent cher, et sont à l'origine de nouveaux défauts.

- il est rarement possible de réutiliser un logiciel existant pour en faire un nouveau produit de remplacement

POURQUOI LA QUALITÉ ?

Selon une étude réalisée par le Standish Group (1994)

- 53 % des logiciels créés sont une réussite mitigée : le logiciel est opérationnel mais le délai de livraison n'a pas été respecté, les budgets n'ont pas été tenus, et certaines fonctionnalités ne sont pas disponibles.
- Le dépassement des coûts est en moyenne de 90 %
- Le dépassement des délais est de 120 %
- La qualité moyenne est estimée à 60 %

POURQUOI LA QUALITÉ ?

Rapport GAO (Government Accountability Office) 2016

- 75 % du budget IT consacré à la maintenance d'anciens systèmes durant l'année 2015
- Depuis 2010, le nombre de projets relatifs à l'exploitation et à la maintenance n'a cessé de croître
- le budget destiné à la modernisation, au développement et à l'amélioration des systèmes déjà existants est en baisse de près de 7,3 milliards de dollars

LA QUALITÉ LOGICIELLE AUJOURD'HUI

- Mise en place de tests unitaires
- Mise en place de règles de programmation
- Mise en place de métriques liées à l'analyse du code
- Mise en pratique et validation sur une plate-forme d'intégration continue

LES LIMITES DE LA QUALITÉ LOGICIELLE AUJOURD'HUI

- La mise en place des tests est considérée comme une perte de temps
- Le respect des normes et métriques se heurte au délais de livraison
- Conséquence : augmentation de la dette technique

PYTHON :

DOCUMENTATION

POURQUOI DOCUMENTER ?

- Informer de ce que fait le code
- Informer pourquoi le code est écrit de cette manière
- Informer sur le comportement du code (des fonctions, objets…)

DEUX OUTILS

- Les commentaires
- Les docstrings

LES COMMENTAIRES

- Un commentaire commence par un croisillon #
- Un commentaire peut être placé n'importe où dans le code
- Un commentaire doit expliquer pourquoi le code suivant a été écrit de cette manière.
- Privilégiez la clarté du code à la présence de commentaires

LES COMMENTAIRES

```
# x is set to 10
x = 10

# x is set to the last list element
x = ma_liste[-1]

# account number is the last element of bank infos
numero_compte = infos_bancaire[-1]
```

LES DOCSTRINGS

- Chaines de caractères encadrés d'un """triple double quotes"""
- Placé à des endroits spécifiques :
  - au début du package
  - après la déclaration d'une classe
  - après la déclaration d'une méthode ou fonction
- Conventions spécifiés dans la PEP 257
  https://www.python.org/dev/peps/pep-0257/

LES DOCSTRINGS

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""Docstring de mon module"""

class MaClasse:
    """Docstring de ma classe"""
    pass

def ma_fonction():
    """Docstring de ma fonction"""
    pass
```

LES DOCSTRINGS

- Les fonctions help() et __doc__ permettent leur affichage
- Les IDEs permettent leur affichage sans consulter le code
- Permettent la génération de documentations (Pydoc, Doxygen, Sphynx)
- Permettent l'illustration par les doctests
- Les doctstrings doivent informer sur comment utiliser le package, classe ou fonction.

LES DOCSTRINGS

- Les outils de génération de doc supportent le format RST
- Exemple de comment documenter
  http://thomas-cokelaer.info/tutorials/sphinx/docstring_python.html

LES DOCSTRINGS

```
def add(a, b):
    """
        Adds two numbers and returns the result.

        :param a: The first number to add
        :param b: The second number to add
        :type a: int
        :type b: int
        :return: The result of the addition
        :rtype: int

        .. seealso:: sub(), div(), mul()
        .. warnings:: This is a completly useless function. Use it only in a
                      tutorial unless you want to look like a fool.
    """
    return a + b
```

TODO

- Il existe une balise .. todo::
- Ne pas utiliser, préférer la convention de commentaire
  # TODO: un truc à faire

LIMITE DE LA DOCUMENTATION

- Maintenance
- Lorsque le code évolue, la documentation doit évoluer
- Aucun outil ne permet de valider la fiabilité d'une documentation

PYTHON : LES TESTS

POURQUOI TESTER ?

- Montrer que le code fonctionne
- Montrer que le code répond aux attentes
- Illustrer l'usage du code
- Montrer que le code fonctionne toujours

LES TESTS EN PYTHON

- Les Doctests
- Le module unittest

DOCTEST

- Tests intégrés à la documentation
- Doctest recherche tous les tests dans le module indiqué et teste le résultat
- Par défaut, affiche les tests échoués

DOCTEST

- Documentation : https://docs.python.org/2/library/doctest.html
- Importer le module doctest
- Utiliser la fonction doctest.testmod()

DOCTEST

```
def add(a, b):
    """
        :Example:

        >>> add(1, 1)
        2
        >>> add(2.1, 3.4)
        5.5

    """
    return a + b

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

UNITTEST

- Automatisation des tests
- Fonctions d'initialisation et finalisation
- Agrégation
- Indépendance des tests

NOTIONS DE TESTS UNITAIRES ET TESTS D'INTÉGRATION

- Un test unitaire doit tester une fonctionnalité et une seule
- Un test unitaire doit être indépendant et isolé du système

UNITTEST

- Importer le module unittest
- Créer une classe héritant de unittest.TestCase par cas testé
- Les méthodes de test doivent commencer par test
- Les méthodes setUp() et tearDown() sont exécutées avant et après chaque test.

UNITTEST

```python
import unittest

from piles import Lifo

class TestCreateLifo(unittest.TestCase):

    def testEmptyArg(self):
        pile = Lifo()
        self.assertEqual(0, len(pile._pile))

    def testFewArg(self):
        pile = Lifo(2, 45)
        self.assertEqual(2, len(pile._pile))
```

UNITTEST

```python
import unittest

from piles import Lifo

class TestLifoEmpile(unittest.TestCase):

    def setUp(self):
        self.pile = Lifo('Han', 'Leia')

    def testEpilerUnElementListe(self):
        self.pile.empile(['Luke', 'Yoda'])
        self.assertEqual(3, len(self.pile._pile))

    def tearDown(self):
        del(self.pile)
```

UNITTEST

assertEqual(a, b)	a == b
assertNotEqual(a, b)	a != b
assertTrue(x)	bool(x) is True
assertFalse(x)	bool(x) is False
assertIs(a, b)	a is b
assertIsNot(a, b)	a is not b
assertIsNone(x)	x is None
assertIsNotNone(x)	x is not None
assertIn(a, b)	a in b
assertNotIn(a, b)	a not in b
assertIsInstance(a, b)	isinstance(a, b)
assertNotIsInstance(a, b)	not isinstance(a, b)

UNITTEST, TESTER LES EXCEPTIONS

```
class MyTestCase(unittest.TestCase):
    def testException(self):
        with self.assertRaises(SomeException) as context:
            broken_function()

        self.assertTrue('This is broken' in context.exception)
```

UNITTEST, POUR LES DIVISIONS

```
class MyTestCase(unittest.TestCase):
    def testDivision(self):
        self.assertAlmostEqual(division(1., 3), 0.3333, 4)
```

AUTRES OUTILS UTILES

- TestSuite permet de grouper des tests pour leur exécution
- Python 2.7 permet de découvrir les tests dans une arborescence

```
python -m unittest discover
```

PYTHON : AUTRES OUTILS QUALITÉ

DÉBOGUEUR

- Intégré à l'IDE
- PDB intégré à Python et s'exécute en ligne de commande
  http://docs.python.org/library/pdb.html
  python -m pdb monfichier.py
- PDB permet l'exploration post-mortem
  import pdb
  pdb.set_trace()

DÉBOGUEUR

PDB : liste commandes

- l : (list) liste quelques lignes de code avant et après
- n : (next) exécute ligne suivante
- s : (step in) entre dans la fonction
- r : (return) sort de la fonction
- unt : (until) si dernière ligne boucle, reprend jusqu'à l'exécution boucle
- q : (quit) quite brutalement le programme
- c : (continue) reprend l'exécution

PYLINT

- Outil d'analyse de code
- Documentation:
  https://www.pylint.org/
- Donne une note sur 10 en fonction de divers critères
- Execution
  pylint monmodule.py
- Existe GUI
  pylint-gui

PYLINT

- (C) convention, violation des standards de programmation
- (R) refactor, mauvaise utilisation du code
- (W) warning, problems spécifique a python
- (E) error, dû à des bugs dans le code
- (F) fatal, une erreur qui a causé l’arret de pylint

PROFILING

- Analyse pour mesurer le temps d'exécution de votre programme

- cProfile (et profile) sont disponibles avec Python

- Voir : https://docs.python.org/2/library/profile.html

- Exemple d'exécution :
  python -m cProfile -s cumtime my_script.py

PYTHON : MANIPULER LES FICHIERS

OUVRIR UN FICHIER

- Le fichier est un type en Python
- La fonction open pour ouvrir un fichier est une fonction de base

OUVRIR UN FICHIER

open a pour paramètres :

- Le chemin du fichier
- Le mode d'ouverture
  - 'r' : lecture seule
  - 'w' : écriture, écrase un fichier existant, crée un fichier inexistant
  - 'a' : écriture en mode ajout, écrit en fin de fichier, le crée si inexistant
  - 'b' : ajouté aux précédents permet l'ouverture en mode binaire

OUVRIR UN FICHIER

```
In [1]: mon_fichier = open('fichier_test.txt', 'r')
```

```
In [2]: mon_fichier
Out[2]: <open file 'fichier_test.txt', mode 'r' at 0x1107b2300>
```

FERMER UN FICHIER

- Toujours fermer un fichier lorsque plus nécessaire
- mon_fichier.close()

MANIPULER UN FICHIER AVEC UN CONTEXT MANAGER

```
>>> with open('fichier.txt', 'r') as mon_fichier:
        texte = mon_fichier.read()
```

LIRE ET ÉCRIRE DANS UN FICHIER

```
>>> mon_fichier = open('fichier.txt', 'r')
>>> texte = mon_fichier.read()
>>> mon_fichier.close()
>>>
>>> mon_fichier = open('fichier.txt', 'w')
>>> mon_fichier.write("first writing test")
18
>>> mon_fichier.close()
>>> mon_fichier.closed
True
```

LIRE ET ÉCRIRE DANS UN FICHIER

- read : retourne le fichier comme une chaine de caractères
- readline : retourne une ligne comme une chaine de caractères
- readlines : retourne le fichier comme une liste de chaines de caractères
- write(str) : écrit le contenu en paramètre
- writelines(sequence) : n'ajoute pas de saut de ligne

MANIPULER LE CURSEUR

- tell : indique la position dans le fichier
- seek(offset[, whence]) : déplace le curseur à une position donnée en fonction du paramètre whence

- os.SEEK_SET ou 0 : position absolue, défaut
- os.SEEK_CUR ou 1 : position courante du curseur
- os.SEEK_END ou 2 : position de la fin

MANIPULER LE CURSEUR

"contenu de mon fichier"

```
>>> import os
>>> f.read(5)
'conte'
>>> f.tell()
5
>>> f.seek(-7, os.SEEK_END)
>>> f.read()
'fichier'
```

ITÉRER SUR UN FICHIER

- Un objet de type file possède un itérateur
- next() : retourne la ligne suivante
- for line in f : itère sur le fichier

LE MODULE PICKLE

- Permet d'enregistrer des données en conservant leur type
- Fonctions dump et load
- Objets Pickler et Unpickler

LE MODULE PICKLE

```
pile_name = "ma pile"
pile_value = [42, 'answer']

import pickle

f = open("pile_backup", w)
pickle.dump(pile_name, f)
pickle.dump(pile_value, f)
f.close
```

```
import pickle

f = open("pile_backup", r)
pile_name = pickle.load(f)
pile_value = pickle.load(f)
f.close
```

MANIPULER LES RÉPERTOIRES

Utiliser le module os

- os.mkdir(chemin, mode) : crée répertoire, mode UNIX
- os.remove(chemin) : supprime fichier
- os.removedirs(chemin) : supprime répertoires récursivement
- os.rename(chemin_old, chemin_new) : renomme fichier ou répertoire
- os.renames(chemin_old, chemin_new) : renomme fichier ou répertoire en créant les répertoires si ils n'existent pas

MANIPULER LES RÉPERTOIRES

Utiliser le module os

- os.chdir(chemin) : change le répertoire de travail
- os.getcwd() : affiche répertoire courant

MANIPULER LES RÉPERTOIRES

Utiliser le module os

- os.path.exists(chemin) : est-ce que le fichier ou répertoire existe
- os.path.isdir(chemin) : est-ce un répertoire
- os.path.isfile(chemin) : est-ce un fichier

MANIPULER LES RÉPERTOIRES

Utiliser le module os

- os.listdir(chemin) : liste un répertoire

Utiliser le module glob qui permet l'utilisation de wildcards

- glob.glob(pattern) : liste le contenu du répertoire en fonction du pattern

Pour glob, les fichiers commençant par un . sont spéciaux et restent par défaut cachés.

MANIPULER LES RÉPERTOIRES

```
>>> import os, glob
>>>
>>> os.listdir(.)
```

```
>>> glob.glob('*')
```

```
>>> glob.glob('./exo[0-9].py')
```

MANIPULER LES RÉPERTOIRES

Actions sur les fichiers et répertoires

- shutil.move(src, dest) : déplace ou renomme un fichier ou un répertoire
- shutil.copy(src, dest) : copie un fichier ou un répertoire
- shutil.copy2(src, dest) : copie un fichier ou un répertoire avec les métadonnées
- os.chmod(path, mode) : change les permissions

MANIPULER LES NOM DE FICHIERS

- os.path.dirname(path) : retourne l'arborescence de répertoires
- os.path.basename(path) : retourne le nom du fichier
- os.path.split(path) : retourne un tuple des deux précédents
- os.path.splitext(path) : retourne un tuple pour obtenir l'extension

PYTHON : ACCÈS AUX BASES DE DONNÉES

ACCÈS AUX BASES RELATIONNELLES

Principe général

- Établir une connexion
- Créer un curseur et lui attribuer une requête
- Exécuter la requête
- Itérer sur les éléments retournés
- Fermer la connexion

ACCÉDER À MYSQL

- Nécessite le pilote MySQLdb
- Disponible via pip ou installer
- Basé sur l'API C de MySQL

ACCÉDER À MYSQL

```
import MySQLdb

try:
    conn = MySQLdb.connect(host='localhost', user='test user',
                           passed='test pass', db='test')
    cursor = conn.cursor()
    cursor.execute("SELECT VERSION()")
    row = cursor.fetchone()
    print 'server version', row[0]

finally:
    if conn:
        conn.close()
```

ACCÉDER À SQLITE

- Base de données fichier
- SQLite ne gère pas d'utilisateurs ni des bases de données
- La stdlib fournit le module sqlite3
- Lors de la connexion, si la base n'existe pas, elle est créée
- On peut travailler avec une base SQLite en mémoire

ACCÉDER À SQLITE

```
import sqlite3 as lite

con = lite.connect('testdb.db')

with con:
    cursor = con.cursor()
    cursor.execute('SELECT SQLITE_VERSION()')
    row = cursor.fetchone()
    print 'server version', row[0]
```

CRÉER UNE TABLE DANS SQLITE

```
con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("CREATE TABLE people (name_last, age)")
```

LES REQUÊTES

- Les requêtes s'exécutent sur un curseur
- Une requête peut être paramétrée
  - avec le caractère ?
  - avec une étiquette

EXÉCUTER UNE REQUÊTE

```
who = "Yeltsin"
age = 72
cur.execute("INSERT INTO people VALUES (?, ?)", (who, age))
```

EXÉCUTER UNE REQUÊTE

```
who = "Yeltsin"
age = 72
cur.execute("select * from people where name_last=:who and age=:age",
            {"who": who, "age": age})
```

RÉCUPÉRER DES ENREGISTREMENTS

- fetchone() : retourne l'enregistrement suivant ou None
- fetchmany([size]) : retourne une liste d'enregistrements
- fetchall() : retourne une liste d'enregistrements

TRANSACTIONS

- Exécuter un commit sur la connexion après avoir exécuté les différentes instructions
  conn.commit()

- Exécuter un rollback dans les exceptions
  conn.rollback()

- Le context manager gère commit et rollback

PYTHON : MANIPULER LES DATES

MANIPULER LES DATES

Python propose plusieurs modules

- time
- calendar
- datetime

DATETIME

- Permet de manipuler les dates sous forme d'objets
- Constructeur :
  datetime(annee, mois, jour, heure, minute, seconde, microseconde, fuseau horaire)
- Seuls année, mois et jour sont obligatoires

DATETIME

```
>>> from datetime import datetime
>>> datetime(2015, 12, 16)
datetime.datetime(2015, 12, 16, 0, 0)
```

DATETIME DATE ACTUELLE

```
>>> from datetime import datetime
>>> datetime.now()
datetime.datetime(2015, 12, 16, 13, 30, 00, 437881)
```

DATETIME ACCÈS VALEURS

```
>>> maintenant.year
2015
>>> maintenant.month
12
>>> maintenant.day
16
>>> maintenant.hour
13
>>> maintenant.minute
30
>>> maintenant.second
00
>>> maintenant.microsecond
437881
>>> maintenant.isocalendar() # année, semaine, jour
(2015, 51, 3)
```

DATETIME MANIPULER LES DATES OU LES HEURES

```
>>> from datetime import date, time, datetime
>>> maintenant = datetime.now()
>>> maintenant.date()
datetime.date(2015, 12, 16)
>>> maintenant.time()
datetime.time(13, 30, 00, 437881)
```

LES DURÉES

- Manipuler les dates c'est manipuler des différences entre deux dates
- En python, représenté par l'objet datetime.timedelta

LES DURÉES

```
>>> duree = datetime(2017, 12, 15) - datetime(2015, 12, 16)
>>> durée
datetime.timedelta(730)
```

LES DURÉES

```
>>> duree = datetime(2017, 12, 15) - datetime.now()
>>> durée
datetime.timedelta(590, 21791, 995830)
```

LES DURÉES

```
>>> duree = datetime(2017, 12, 15) - datetime.now()
>>> durée
datetime.timedelta(590, 21791, 995830)
>>> duree.days
590
>>> duree.seconds
21791
>>> duree.microseconds
995830
```

LES DURÉES

```
>>> cuisson_oeuf = timedelta(seconds=180)
>>> datetime.now() + cuisson_oeuf
datetime.datetime(2015, 12, 16, 13, 33, 00, 995830)
```

DATETIME ET TIMEDELTA

- Les objets datetime et timedelta sont immuables
- À chaque modification, on obtient un nouvel objet
  maintenant.replace(year=1995) retourne un nouvel objet

FORMATER LES DATES

- Définir un format sous forme d'une chaine de caractères
- Voir https://docs.python.org/2/library/time.html#time.strftime
- Formatage appliqué par strftime

LES DURÉES

```
>>> import datetime
>>> datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S.%f')
'2015-12-16 13:30:995830'
```

MODULE CALENDAR

Module permettant de manipuler et interroger un calendrier

MODULE CALENDAR

```
>>> import calendar
>>> calendar.mdays # combien de jour par mois ?
[0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
>>> calendar.isleap(2000) # est-ce une année bissextile ?
True
>>> calendar.weekday(2000, 1, 1) # quel jour était cette date ?
5
>>> calendar.MONDAY, calendar.TUESDAY, calendar.WEDNESDAY
(0, 1, 2)
```

PYTHON : REGEX

QU'EST-CE QUE LES REGEX ?

- Les expressions régulières sont un puissant moyen de rechercher et d'isoler des expressions d'une chaîne de caractères.

- Permet de rechercher un motif
- Permet de valider une chaine de caractères
- En python, fournis par le module re
  https://docs.python.org/2/library/re.html

DÉFINIR UN MOTIF

- Une chaine de caractères
- Marqueurs pour indiquer certaines caractéristiques
- 'chat' permet de rechercher le motif 'chat' et le trouvera 'chaton' ou 'achat'

DÉFINIR UN MOTIF : DÉBUT OU FIN DE CHAINE

- ^ précédant le motif impose de trouver le motif en début de chaine.
  '^chat' reconnait 'chaton' mais pas 'achat'

- $ suivant le motif impose de trouver le motif en fin de chaine.
  'chat$' reconnait 'achat' mais pas 'chaton'

DÉFINIR UN MOTIF : NOMBRE D'OCCURRENCES

Quand un caractère est suivi de ces symboles, on recherche

- * : 0, 1 ou plus d'occurrences
- + : 1 ou plus d'occurrences
- ? : 0 ou 1 occurence
- {m} : m occurrences exactement
- {m, n} : de m à n occurrences
- {, n} : 0 à n occurrences
- {m,} : au moins m occurrences

DÉFINIR UN MOTIF : CLASSES DE CARACTÈRES

L'usage de crochets permet de définir les caractères acceptés

- [abcd] : un caractère parmi a, b, c et d
- [a-d] : un caractère entre a et d
- [a-d, A-D] : un caractère entre a et d en minuscule ou capitale
- [^a-d] : n'importe quel caractère sauf ceux entre a et d
- . : signifie n'importe quel caractère

DÉFINIR UN MOTIF : LES GROUPES

Les parenthèses permettent de définir des groupes de caractères

- (tcha){3} : est équivalent à 'tchatchatcha'
- (Android|ios) : permet de valider 'Android' ou 'ios'

UTILISER LE MODULE RE

- Dans une séquence pour les expressions régulières, les caractères spéciaux doivent être échappés par un \
  '\\n'
- Les caractères spéciaux seront échappés si la chaine est précédée par un r
  r'\n'

UTILISER LE MODULE RE : SEARCH

- re.search(pattern, seq) recherche le pattern dans ses
- Si le motif est trouvé, retourne un objet de type _sre.SRE_Match
- Si le motif n'est pas trouvé, retourne None
- re.match est similaire à search mais recherche en début de chaine

UTILISER LE MODULE RE : SEARCH

```
>>> re.search('chat', 'un chat')
<_sre.SRE_Match at 0x11099e510>
>>> re.search('chat', 'chaton')
<_sre.SRE_Match at 0x11099e578>
>>> re.search('chat', 'achat')
<_sre.SRE_Match at 0x11099e5e0>
>>> re.search('chat', 'vente')
```

UTILISER LE MODULE RE : UTILISER UNE REGEX COMME UN OBJET

```
chat_regex = re.compile('^chat')
>>> chat_regex.search('un chat')
>>> chat_regex.search('chaton')
< sre.SRE_Match at 0x11099e648>

>>> type(chat_regex)
_sre.SRE_Pattern
```

UTILISER LE MODULE RE : AFFICHER LE RÉSULTAT

Sur un MatchObject

- group permet d'afficher le motif trouvé
- group(i) permet d'afficher le motif (si i vaut 0) ou sous groupe définit par des parenthèses
- start renvoi l'indice de début
- end renvoi l'indice de fin

UTILISER LE MODULE RE : AFFICHER LE RÉSULTAT

```
>>> float_regex = re.compile('([0-9]+)\.([0-9]+)')
>>> ma_chaine = 'pi vaut 3.14'
>>> resultat = float_regex.search(ma_chaine)
>>> resultat.group()
'3.14'
>>> resultat.group(0)
'3.14'
>>> resultat.group(1)
'3'
>>> resultat.group(2)
'14'
>>> resultat.start()
8
>>> resultat.end()
12
```

UTILISER LE MODULE RE : TROUVER TOUTES LES OCCURRENCES

- search ne renvoi que la première occurence
- findall renvoi une liste de toutes les occurrences
- finditer est un itérateur fonctionnant comme findall

UTILISER LE MODULE RE :
TROUVER TOUTES LES OCCURRENCES

```
>>> float_regex = re.compile('[0-9]+\.[0-9]+')
>>> ma_chaine = 'pi vaut 3.14 et e vaut 2.72'
>>> resultat = float_regex.findall(ma_chaine)
>>> resultat
['3.14', '2.72']
```

UTILISER LE MODULE RE : SUBSTITUTIONS

```
>>> float_regex = re.compile('[0-9]+\.[0-9]+')
>>> ma_chaine = 'pi vaut 3.14 et e vaut 2.72'
>>> float_regex.sub('quelque chose', ma_chaine)
'pi vaut quelque chose et e vaut quelque chose'
>>> float_regex.sub('quelque chose', ma_chaine, count=1)
'pi vaut quelque chose et e vaut 2.72'
```

PYTHON :

DÉCORATEURS

BASE DE LA MÉTAPROGRAMMATION

- Métaprogrammation : désigne l'écriture de programmes qui manipulent des données décrivant elles-mêmes des programmes
- Décorateurs : fonctions qui vont modifier le comportement d'autres fonctions ou classes

UTILISATION

```
@nom_du_decorateur
def ma_fonction():
    pass
```

FONCTIONS : RAPPEL

Une fonction peut être assignée à une variable

```
>>> def crier(mot="yes"):
        return mot.capitalize() + "!"

>>> print(crier())
'Yes!'
>>> hurler = crier
>>> print(hurler())
'Yes!'
```

FONCTIONS : RAPPEL

Une fonction peut être définie dans une autre fonction

```
>>> def parler():
        def chuchoter(mot="yes"):
            return mot.lower()+"..."
        print(chuchoter())

>>> parler()
'yes...'
```

FONCTIONS : RAPPEL

- Une fonction peut être assignée à une variable
- Une fonction peut être définie dans une autre fonction
- Une fonction peut être passée en paramètre d'une autre fonction
- Une fonction peut être un paramètre de retour d'une autre fonction

DÉCORATEUR

Une fonction qui prend en paramètre une fonction pour exécuter du code avant et/ou après l'appel de cette fonction.

DÉCORATEUR

```
>>> def essai_decorateur(fonction_a_decorer):
        def wrapper_autour_de_la_fonction_originale():
            print("Avant que la fonction ne s’exécute")
            fonction_a_decorer()
            print("Après que la fonction se soit exécutée")
    return wrapper_autour_de_la_fonction_originale
```

DÉCORATEUR

```
>>> @essai_decorateur
>>> def ma_fonction:
        print("dans ma fonction")

>>> ma_fonction()
'Avant que la fonction ne s’exécute'
'dans ma fonction'
'Après que la fonction se soit exécutée'
```

DÉCORATEUR

```
>>> essai_decorateur(ma_fonction)
```

CONCLUSION