

DARKO STANKOVSKI

FORMATION PYTHON

- ▶ Introduction
- ▶ La boite à outils Python
- ▶ Les bases
- ▶ La programmation orientée objet
- ▶ La stdlib
- ▶ Qualité
- ▶ Web
- ▶ Interfaces graphiques

DARKO STANKOVSKI

PYTHON : INTRODUCTION

IT WAS NICE TO LEARN
PYTHON;
A NICE AFTERNOON

D. Knuth, Trento, 2012

HISTORIQUE

- ▶ Crée 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 (pipeline et front)
- ▶ 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

- ▶ CPython : interpréteur de référence en C
- ▶ Autres interpréteurs : Jython (Java), IronPython (.Net)
- ▶ Performance : PyPy (Attention, certaines incompatibilités)
- ▶ Runtime Grumpy pour Youtube :
<https://opensource.googleblog.com/2017/01/grumpy-going-python.html>

CARACTÉRISTIQUES DU LANGAGE

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

PYTHON 2 OU PYTHON 3 ?

Python 2.7 est régulièrement choisi car...

- ▶ Existant encore fortement en Python 2.x
- ▶ Existe encore des dépendances en Python 2.x

PYTHON 2 OU PYTHON 3 ?

- ▶ Support Python 2 prend fin en 2020
- ▶ Depuis 2010 (2.7), Python 2 ne reçoit plus d'évolution
- ▶ Depuis mai 2015, maintenance Python 2 irrégulière
<https://hg.python.org/peps/rev/76d43e52d978>
- ▶ Nouveau projet doit reposer sur Python 3
- ▶ Transition facilitée par le package **future**

DARKO STANKOVSKI

PYTHON : BOITE À OUTILS

LES OUTILS NÉCESSAIRES

- ▶ Interpréteur
- ▶ Gestionnaire de packages
- ▶ iPython
- ▶ Virtualenv et outils associés
- ▶ Environnement de développement
- ▶ Portabilité du code Python 2

OUTIL INDISPENSABLE : L'INTERPRÉTEUR

- ▶ Linux : Python présent, version dépend de la distribution
- ▶ Os X : Python 2.7 installé par défaut
- ▶ Installation :
<https://www.python.org/downloads/>
- ▶ Installation Linux : privilégier le gestionnaires de paquets

L'INTERPRÉTEUR INTERACTIF

- ▶ Dans un terminal, lancer Python par la commande **python**



```
[Thor-2:~ dad3zero$ python
Python 2.7.10 (default, Oct 23 2015, 19:19:21)
[GCC 4.2.1 Compatible Apple LLVM 7.0.0 (clang-700.0.59.5)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> ]
```

LE HELLO WORLD

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

GESTIONNAIRE DE PACKAGES : PIP

- ▶ Installé dans certains cas (version, packages...)
- ▶ 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](http://ipython.org)
- ▶ Informations : <http://ipython.org/>
- ▶ Documentation : <http://ipython.readthedocs.io/>

VIRTUALENV

- ▶ Outil permettant de créer des environnements isolés
- ▶ Permet de contrôler les dépendances et les versions de l'interpréteur
- ▶ Installation : [pip install virtualenv](#)
- ▶ Voir : <https://virtualenv.pypa.io/>

VIRTUALENV : VITUALENVWRAPPER

- ▶ Simplifie l'usage de virtualenv
- ▶ Installation : [pip install virtualenvwrapper](https://virtualenvwrapper.readthedocs.org/)
- ▶ Voir : <https://virtualenvwrapper.readthedocs.org/>

VITUALENVWRAPPER : UTILISATION

- ▶ `mkvirtualenv monEnv` crée un virtualenv
- ▶ `workon monEnv` active le virtualenv
- ▶ `lssitepackages` liste les packages du virtualenv
- ▶ `deactivate` désactive le virtualenv
- ▶ `rmvirtualenv monEnv` supprime le virtualenv

ENVIRONNEMENT DE DÉVELOPPEMENT

Assurez-vous d'être en UTF-8

- ▶ IDLE
 - fourni avec installation standard, codé en Python avec tkinter
- ▶ 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`

OUTIL DE FORMATION : RESSOURCES

Documents annexes de la formation disponibles sur Github :

<https://github.com/darko-itpro/training-python>

Clonez en local le repo afin d'utiliser les Notebooks, docs et le package dédié aux stagiaires

Voir readme.md (ou la page du projet sur Github) pour les instructions d'installation et configuration.

OUTIL DE FORMATION : JUPITER NOTEBOOK

Issu du projet ipython, document contenant du texte riche et du code. Utilisable comme cahier d'exercice.

- ▶ <http://jupyter.org/>
- ▶ `pip install jupyter`
- ▶ Executer un notebook par :
 - ▶ `jupyter notebook`
 - ▶ `jupyter notebook mybook.ipynb`

DARKO STANKOVSKI

PYTHON : LES BASES

PRÉREQUIS

- ▶ Variable
- ▶ Type de donnée
- ▶ Structure de contrôle
- ▶ Fonction
- ▶ Objet

EN PYTHON, TOUT EST OBJET

- ▶ Les objets peuvent avoir des attributs (une donnée)
`monObjet.monAttribut`
- ▶ Les objets peuvent avoir des méthodes (une action sur l'objet)
`monObjet.maMethode()`

DONNÉES ET VARIABLES

- ▶ Les variables référencent des données
- ▶ Les variables contiennent une **référence** vers une valeur
- ▶ Type déterminé dynamiquement (pas de déclaration)
- ▶ Donnée détruite lorsqu'elle n'est plus accessibles
(compteur de références)
- ▶ Règles nommage variables : peut contenir **a-z, A-Z, _ ou 0-9** (sauf premier caractère) (**[a-zA-Z_][a-zA-Z0-9_]***)

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 DES VALEURS

```
>>> 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 valeur d'une variable : `print(var)`
- ▶ Afficher le type de la valeur : `print(type(var))`
- ▶ 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
- ▶ Portabilité assurée par l'instruction
`from __future__ import print_function`

TYPES NUMÉRIQUES

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

TYPES NUMÉRIQUES, ENTIERS

- ▶ Si la valeur d'un entier est comprise entre -2^{n-1} et $2^{n-1}-1$, il est géré en registre
- ▶ Cette valeur maximum est donnée par `sys.maxsize` (taille maximum de conteneurs en Python)

EXEMPLE DE DÉCLARATION DE TYPES

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

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

>>> 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 : `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
$\sim x$	Inversion

OPÉRATEURS BINAIRES, 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, list, tuple, dictionnaires**
- ▶ Chaînes de caractères et tuples sont des séquences immuables

LES SÉQUENCES

```
>>> var_string = 'Characters string'  
>>> var_string = "Characters string"  
>>> var_string = str(42)  
  
>>> var_list = ["Long", "characters", "string"]  
>>> var_list = list(("Long", "characters", "string"))  
  
>>> var_tuple = "Long", "characters", "string"  
>>> var_tuple = ("Long", "characters", "string")  
>>> var_tuple = tuple(("Long", "characters", "string"))  
>>> var_tuple = tuple(["Long", "characters", "string"])  
>>> var_tuple = ("A string", )
```

OPÉRATIONS COMMUNES SUR LES SÉQUENCES (HORS DICT)

<code>x in s</code>	True si s contient x, sinon False
<code>x not in s</code>	False si s contient x, sinon True
<code>s + t</code>	Concaténation
<code>s * n</code>	Répétition
<code>s[i]</code>	Élément à l'indice ou clef i
<code>len(s)</code>	Taille de la chaîne
<code>min(s)</code>	Plus petit élément de la séquence
<code>max(s)</code>	Plus grand élément de la séquence
<code>s.index(x)</code>	Indice de la première occurrence de x
<code>s.count(x)</code>	Nombre total d'occurrences de x

LES CHAINES DE CARACTÈRES

- ▶ Les chaînes de caractère sont délimités par des simple ou double quotes
- ▶ String literals délimités par des triple-double-quotes
- ▶ Sont immuables
- ▶ Sont encodées en unicode
- ▶ En Python 2, chaînes de type **str** sont encodées en ASCII, portabilité assurée par l'instruction
from __future__ import unicode_literals

OPÉRATIONS SPÉCIFIQUES SUR LES CHAINES

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

LES VARIABLES : INTERACTION AVEC L'UTILISATEUR

- ▶ Saisie de variable avec la fonction `input("question ")`
- ▶ Usage :
`name = input('Quel est votre nom ? ')`
- ▶ Input type la saisie comme une chaîne de caractères.
- ▶ Usage :
`age = int(input("quel est votre âge ? "))`

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

- ▶ En Python 2, Input interprète la saisie
- ▶ Existe instruction raw_input se comportant comme le input Python 3
- ▶ Compatibilité Python 3 dans Python 2 par
`from builtins import input`

LES VARIABLES : INTERACTION AVEC L'UTILISATEUR

- ▶ L'affichage sur la console est réalisé par la fonction `print()`
- ▶ Utilisation de chaînes de caractères formatées avec l'opérateur `%` (printf-style) ou la fonction `format`
- ▶ Python 3.6 a ajouté les *string literals*

AFFICHAGE FORMATÉ AVEC L'OPÉRATEUR % (STYLE PRINTF)

- ▶ Syntaxe :

"Ma variable : %type" % var

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

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

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

- ▶ Précision pour les float :

"Resultat: %.2f" % 3.141592653589793"

AFFICHAGE FORMATÉ : EXEMPLES OPÉRATEUR %

```
>>> answer = input("Réponse ? ")
Réponse ? 42
>>> pi = 3.14
>>> print(answer)
42
>>> print("La réponse est %d" % answer)
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).2f" % {'var':var, 'val':pi})
pi est égal à 3.14
```

AFFICHAGE FORMATÉ AVEC LA FONCTION FORMAT

- ▶ Syntaxe : `string.format(*args)`
- ▶ "Résultat : {}".format(var)
- ▶ "Résultat : {}, {}".format(var1, var2)
- ▶ "Résultat : {1} {0}".format(var1, var2)
- ▶ "Résultat : {value} {unit}".format(unit=var1, value=var2)
- ▶ "Résultat : {:.2f}".format(var)
- ▶ "Résultat : {value:.2f} {unit}".format(unit=var1, value=var2)

Voir : <https://docs.python.org/3/library/string.html#formatstrings>

AFFICHAGE FORMATÉ : EXEMPLES FONCTION FORMAT

```
>>> answer = int(raw_input("Réponse ? "))

Réponse ? 42
>>> pi = 3.14
>>> print("La réponse est {}".format(answer))
La réponse est 42
>>> var = "pi"
>>> print("{} est égal à {:.2f}".format(var, pi))
pi est égal à 3.14
>>> print("{} est égal à {:010.2f}".format(var, pi))
pi est égal à 0000003.14
>>> nbr = 158.156658
>>> print("nbr ={:.3e}".format(nbr))
nbr =1.582e+02
```

FORMATTED STRING LITERALS

Permettent d'intégrer une variable dans la chaîne à afficher

```
>>> name = input("What is your name ? ")  
What is your name ? John  
>>> print(f"So, your name is {name}")  
So, your name is John
```

```
>>> pi = 3.141592653589793  
>>> f"π : {pi:{10}.{3}}"  
'π : 3.14'
```

https://docs.python.org/3/reference/lexical_analysis.html#f-strings

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 FONCTIONS

`my_list.append(objet)`

ajoute un objet à la fin de la liste

`my_list.count(valeur)`

compte le nombre d'occurrences d'une valeur

`my_list.extend(séquence)`

ajoute une séquence à la liste

`my_list.index(valeur[, start[, stop]])`

retourne l'index de la première occurrence (start et stop sont optionnel)

`my_list.insert(index, objet)`

ajoute un objet avant l'index

`my_list.pop([index])`

supprime et retourne l'objet à l'index (index optionnel)

`my_list.remove(value)`

supprime la première occurrence

`my_list.reverse()`

reverse la liste

`my_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[0] = '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 `indice > 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 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

<code>my_dict[clef]</code>	Retourne la valeur pour <i>clef</i>
<code>my_dict[clef] = value</code>	ajoute ou modifie une paire
<code>my_dict.clear()</code>	vide le dictionnaire
<code>my_dict.copy()</code>	créer une copie du dictionnaire
<code>my_dict.get(clef [, defaut])</code>	retourne la valeur de la clef ou défaut.
<code>clef in my_dict</code>	retourne True si la clef est dans le dictionnaire
<code>my_dict.items()</code>	retourne une liste de tuple (clef, valeur)
<code>list(my_dict)</code>	retourne la liste des clefs
<code>list(my_dict.values())</code>	retourne la liste des valeurs
<code>my_dict.pop(clef)</code>	supprime la paire et retourne la valeur

PYTHON : LES BASES

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

PRINCIPE



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

!=

Different 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

TYPES BOOLÉENS

~~if mon_tableau != None and len(mon_tableau) > 0:
 code~~

if mon_tableau:
 code

```
graph LR; A[if mon_tableau] --> B[mon_tableau != None]; A --> C[len(mon_tableau) > 0]
```

STRUCTURE CONDITIONNELLE

```
>>> x = int(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

```
>>> status = "retard" if delay < 10 else "annulé"
```

LES BOUCLES AVEC FOR

- ▶ En Python, la boucles for parcourt des séquences
- ▶ En-tête : **for element in sequence:**

LES BOUCLES AVEC FOR

```
In [10]: trainings = ['java', 'python', 'swift']

In [11]: for t in trainings:
...:     print(t, len(t))
...:
java 4
python 6
swift 5
```

LA FONCTION RANGE

- ▶ Générateur d'itérable (listes en Python 2) d'entiers
- ▶ `range(x)`
- ▶ `range(x, y)`
- ▶ `range(x, y, z)`
- ▶ Usage dans les boucles : `for i in range(x):`

LA FONCTION RANGE : EXEMPLES

```
In [1]: print(list(range(10)))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [2]: print(list(range(5, 10)))
[5, 6, 7, 8, 9]

In [3]: print(list(range(10, 30, 2)))
[10, 12, 14, 16, 18, 20, 22, 24, 26, 28]

In [4]: range(10)
Out[4]: range(0, 10)
```

LES BOUCLES AVEC WHILE

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

LES BOUCLES AVEC WHILE

```
In [10]: a, b = 0, 1
In [11]: 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

```
In [10]: sums = []

In [11]: for numbers in range(10):
...:     if numbers % 2 == 0:
...:         sums.append(sum(range(numbers)))
...:

In [12]: print(sums)
[0, 1, 6, 15, 28]
```

COMPREHENSION LISTS

```
In [10]: sums = []

In [11]: for numbers in range(10):
...:     if numbers % 2 == 0:
...:         sums.append(sum(range(numbers)))
...:

In [12]: print(sums)
[0, 1, 6, 15, 28]
```

```
In [10]: print([sum(range(numbers))
...:             for numbers in range(10)
...:             if numbers % 2 == 0])
[0, 1, 6, 15, 28]
```

DARKO STANKOVSKI

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, math.sqrt(9))
3.14159265359 3

In [3]: from math import pi, sqrt
In [4]: print(pi, sqrt)
3.14159265359 3

In [5]: from math import *
In [6]: print(pi, sqrt(9))
3.14159265359 3
```

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 giveAnswer():
    print("It's 42")

giveAnswer()
```

```
nv python
: utf-8 -*-

er():
t's 42)

== '__main__':
er()
```

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 = ['my_module', 'mY_other_module']

STRUCTURE D'UN PROJET

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

STRUCTURE D'UN PROJET

README.rst
LICENSE
setup.py
requirements.txt
myproject/__init__.py
myproject/core.py
myproject/helpers.py
docs/conf.py
docs/index.rst
tests/test_basic.py
tests/test_advanced.py

DISTRIBUTION DES PACKAGES

- ▶ Python fournit un outil pour créer un installateur
- ▶ 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 - SETUP.PY

```
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 une arborescence et adapter le **PYTHONPATH**
 - ▶ En déclarant la variable PYTHONPATH
 - ▶ En utilisant les virtualenvs : add2virtualenv path
- ▶ Utiliser les Submodules Git
- ▶ 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 `importlib.reload(monmodule)`
- ▶ Les instances ne sont pas mises à jour

SHELL INTERACTIF ET MODULES

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

DARKO STANKOVSKI

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
```

PYTHON : LES BASES

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

```
In [10]: def create_account(id, value, overdraft):  
...:     pass  
...:
```

```
In [11]: create_account("TXH1138", 200, False)
```

```
In [12]: def create_account(id, value=100, overdraft=False):  
...:     pass  
...:
```

```
In [13]: create_account("TXH1138", 200, False)
```

```
In [14]: create_account("TXH1138")
```

```
In [15]: create_account("TXH1138", 500)
```

```
In [16]: create_account("TXH1138", overdraft=True)
```

VARIADICS, NOMBRE VARIABLE D'ARGUMENTS

- ▶ 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

```
In [1]: var = 10
In [2]: def func():
...:     var = 11
...:     print(var)
...:
In [3]: func()
11

In [4]: print(var)
10
```

```
In [1]: var = 10
In [2]: def func():
...:     global var
...:     var = 11
...:     print(var)
...:
In [3]: func()
11

In [4]: print(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 éphémères
- ▶ 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-clé `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
...
```

DARKO STANKOVSKI

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

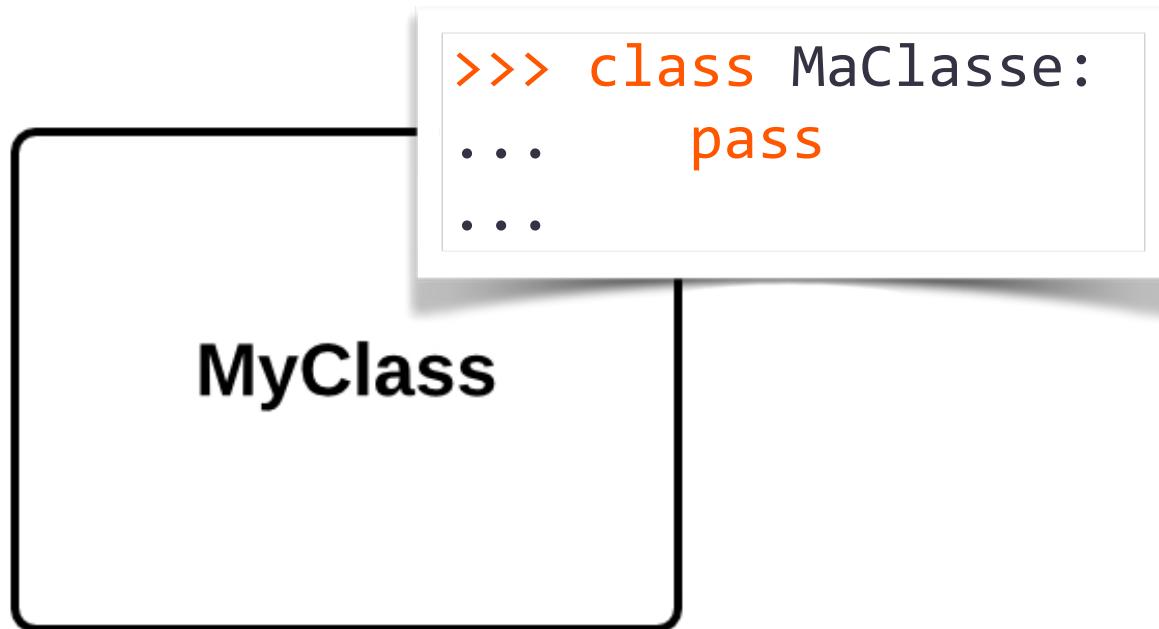
- ▶ Un objet est une instance d'une classe
- ▶ En POO, nous définissons des classes
- ▶ En POO, nous manipulons des instances des classes
- ▶ Le type d'un objet est sa classe

MODÉLISATION ET PRÉSENTATION

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
- ▶ instantiation par le nom de classe suivi de parenthèses

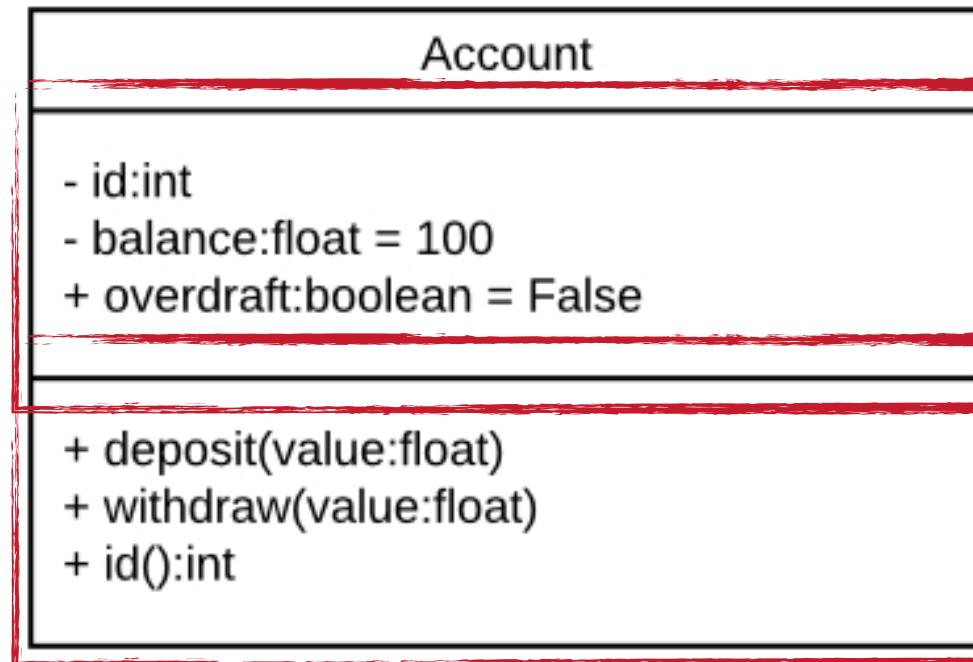
DÉCLARATION

```
>>> class MaClasse:  
...     pass  
...  
>>> maClasse = MaClasse()
```

REPRÉSENTER UNE CLASSE

ATTRIBUTS

MÉTHODES



LES MÉTHODES

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

```
In [54]: class Accout:  
...:     def balance(self):  
...:         print('Wealthy')  
...:  
  
In [56]: myAccount = Accout()  
  
In [57]: myAccount.balance()  
Wealthy
```

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 D'INSTANCE

```
In [10]: class Accout:  
...:     def set_balance(self, value):  
...:         self.balance = value  
...:     def get_balance(self):  
...:         return self.balance  
...:  
  
In [11]: myAccount = Accout()  
  
In [12]: myAccount.set_balance(100)  
  
In [13]: myAccount.get_balance()  
Out[14]: 100
```

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

```
In [10]: class UselessClass:  
....:     def __init__(self):  
....:         pass  
....:  
  
In [20]: class Account:  
....:     def __init__(self, id, balance, overdraft):  
....:         self._id = id  
....:         self._balance = balance  
....:         self.overdraft = overdraft  
....:  
  
In [20]: class Account:  
....:     def __init__(self, id,  
....:                  balance=100,  
....:                  overdraft=False):  
....:         ...
```

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

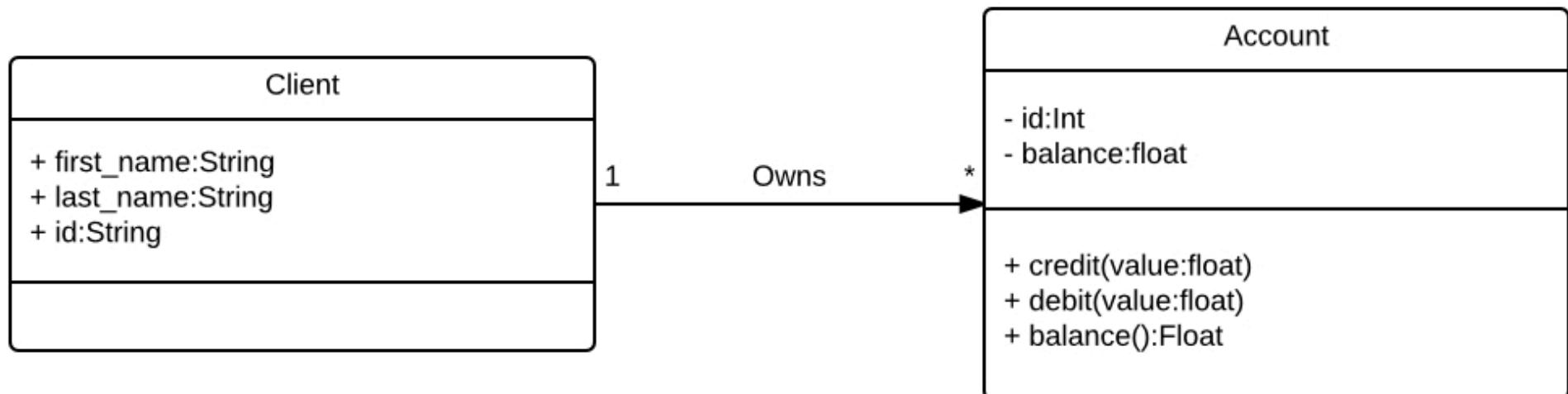
MÉTHODES SPÉCIALES

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

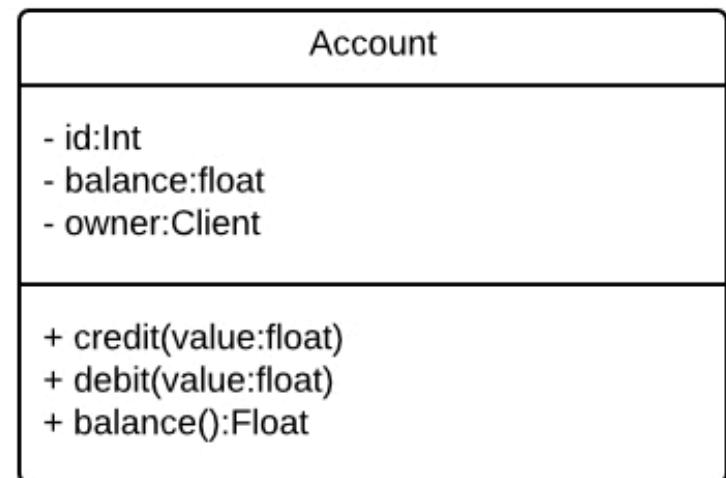
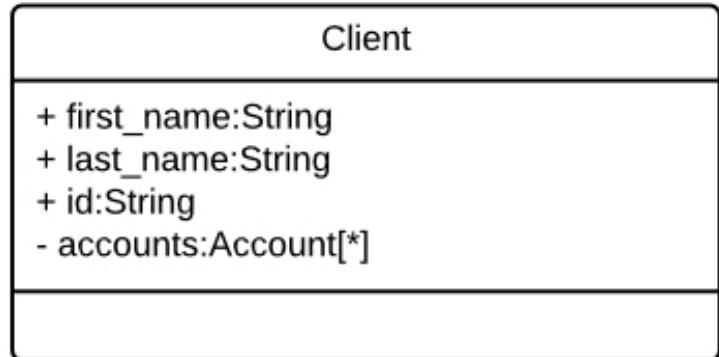
MÉTHODES SPÉCIALES

<code>__neg__</code>	$-x$
<code>__add__</code>	$x + y$
<code>__sub__</code>	$x - y$
<code>__mul__</code>	$x * y$
<code>__div__</code>	x / y

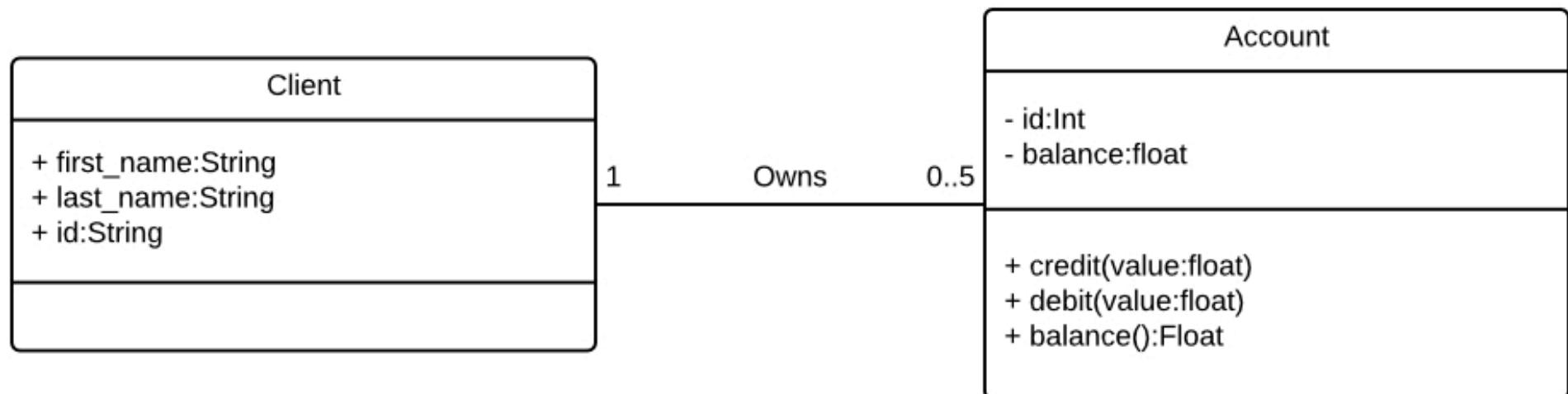
RELATIONS ENTRE CLASSES : ASSOCIATION



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

Compound

```
In [10]: class Compound:  
...:     def __init__(self, amount, temperature, volume):  
...:         ...  
...:         self._volume = volume
```

```
+ temperature():float
```

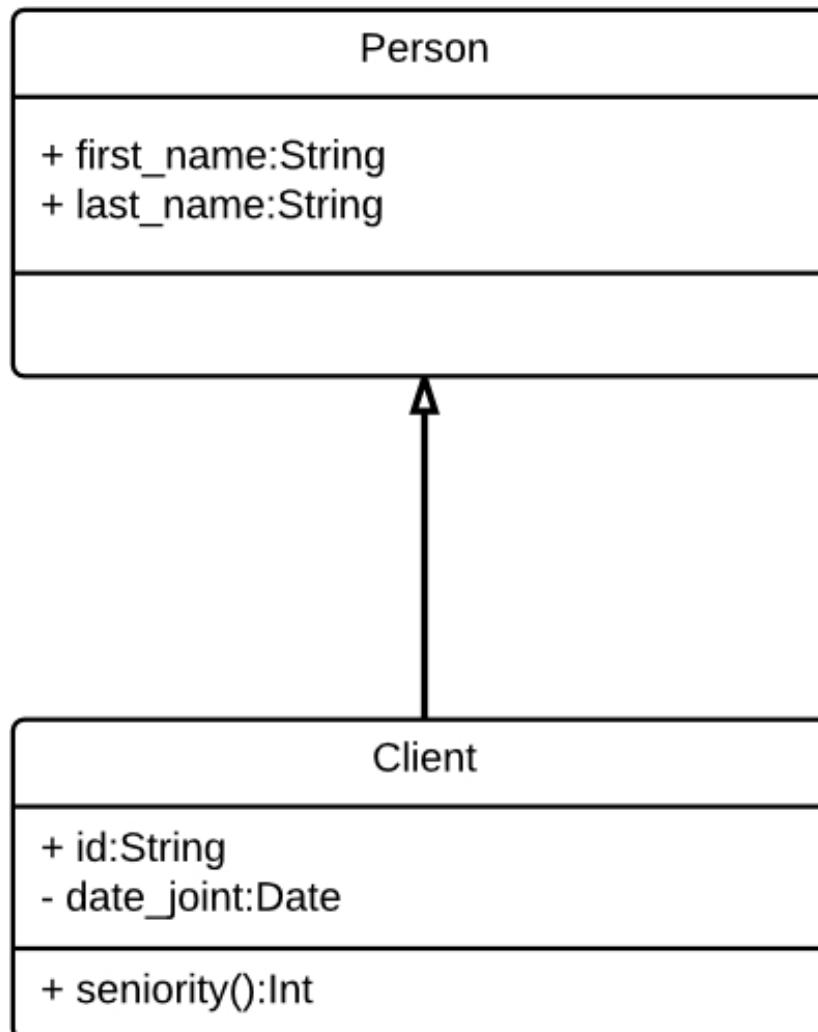
```
def pressure(self):  
    return (self._amount * self._n * self._temperature) / self._volume  
    + heat(energy:float)
```

```
def heat(self, energy):  
    self.temperature += self._heat_cst * energy
```

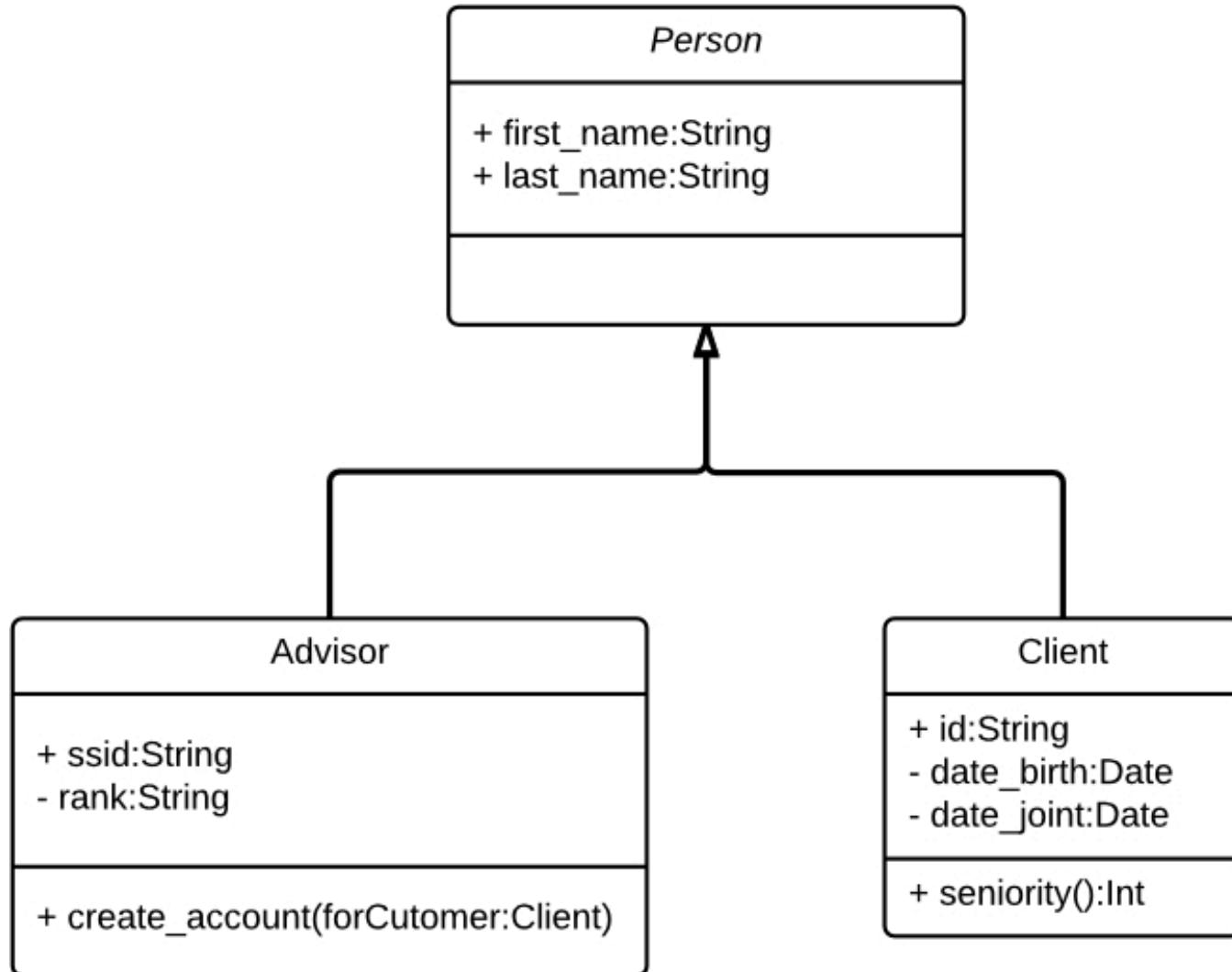
L'HÉRITAGE

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

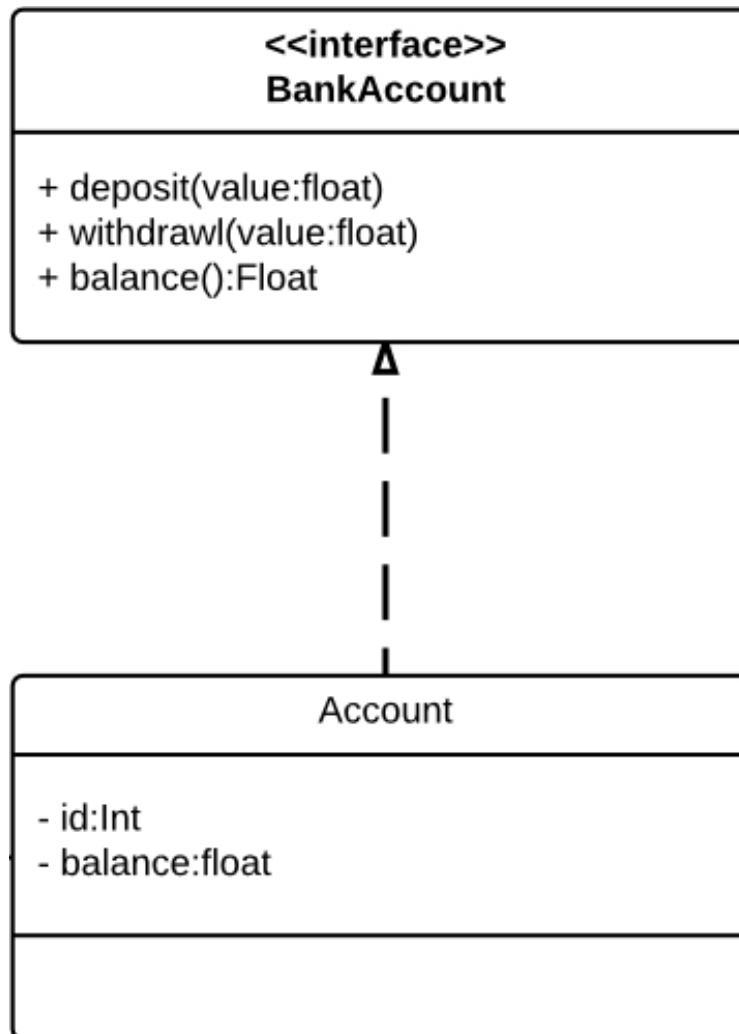
L'HÉRITAGE



L'HÉRITAGE



ABSTRACTION : INTERFACES



L'HÉRITAGE

- ▶ Déclaration de la classe suivi du nom 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

```
In [67]: class Client(Person):
....:     def __init__(self, first_name, last_name, id):
....:         Person.__init__(self, first_name, last_name)
....:         self._id = id
....:         self.join_date = datetime.now()
```

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

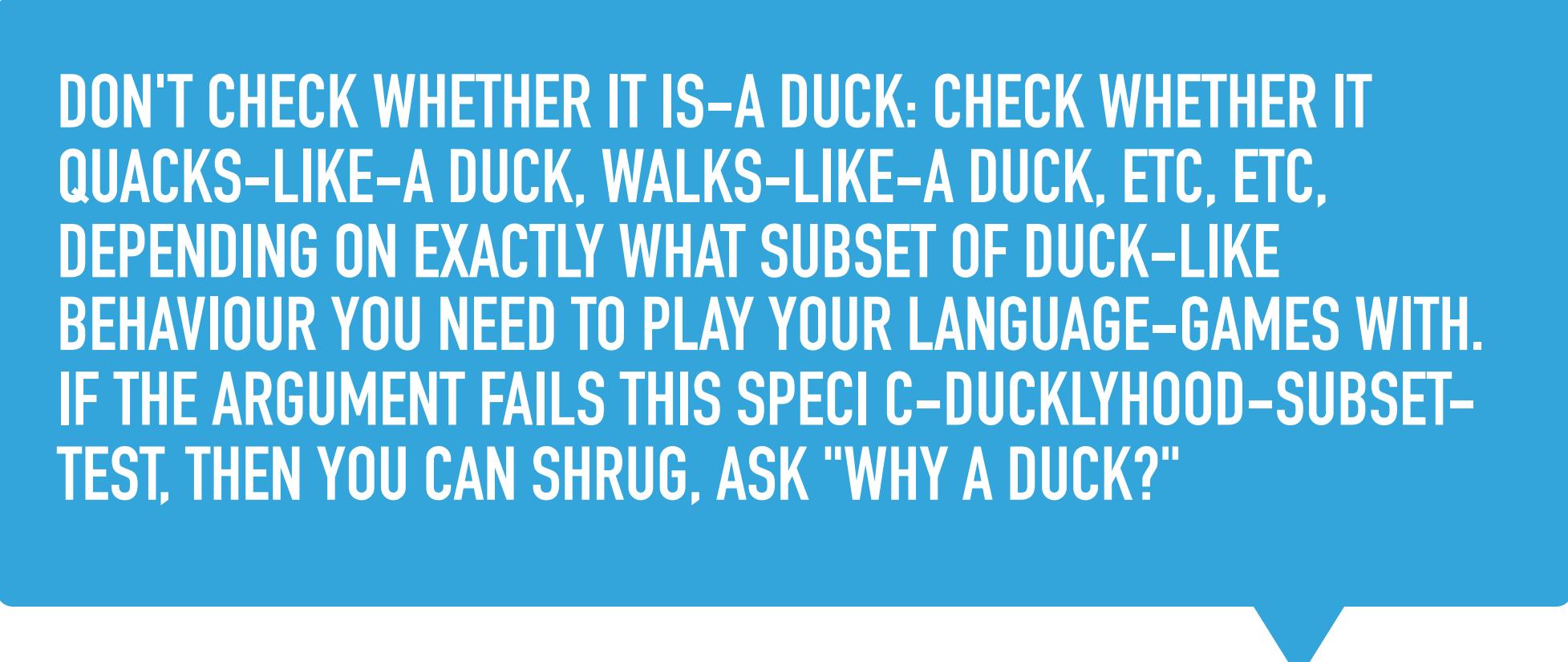
POLYMORPHISME ET DUCK TYPING

- ▶ 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

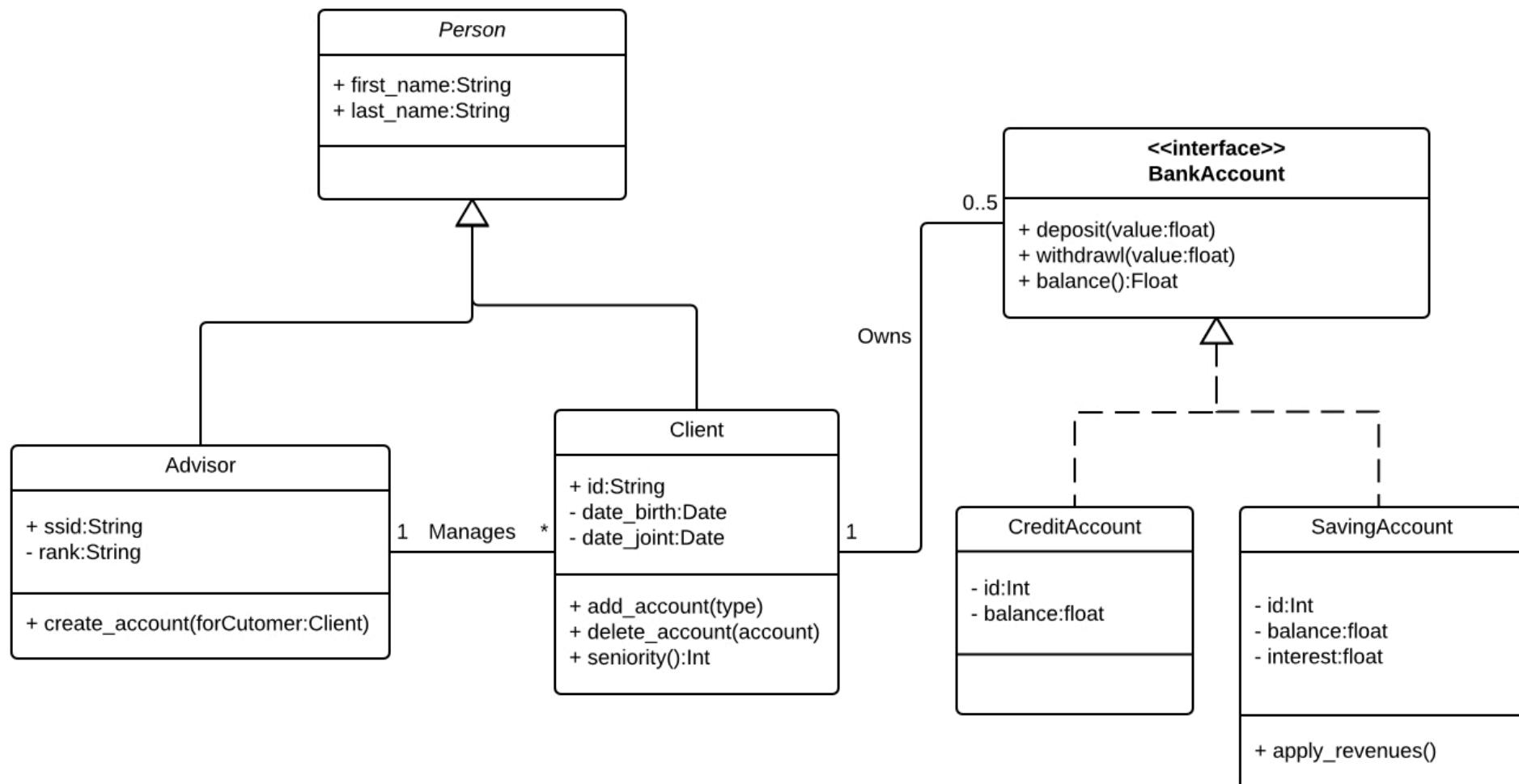
```
>>> def triple_and_one(a, b):
...     print a * 3 + b
...
>>> triple_and_one(5, 2)
17
>>> triple_and_one("hip ", "hora")
hip hip hip hora
```

DON'T CHECK WHETHER IT IS-A DUCK: CHECK WHETHER IT QUACKS-LIKE-A DUCK, WALKS-LIKE-A DUCK, ETC, ETC, DEPENDING ON EXACTLY WHAT SUBSET OF DUCK-LIKE BEHAVIOUR YOU NEED TO PLAY YOUR LANGUAGE-GAMES WITH. IF THE ARGUMENT FAILS THIS SPECI C-DUCKLYHOOD-SUBSET-TEST, THEN YOU CAN SHRUG, ASK "WHY A DUCK?"



Alex Martelli

EXEMPLE COMPLET



CAS D'UTILISATION

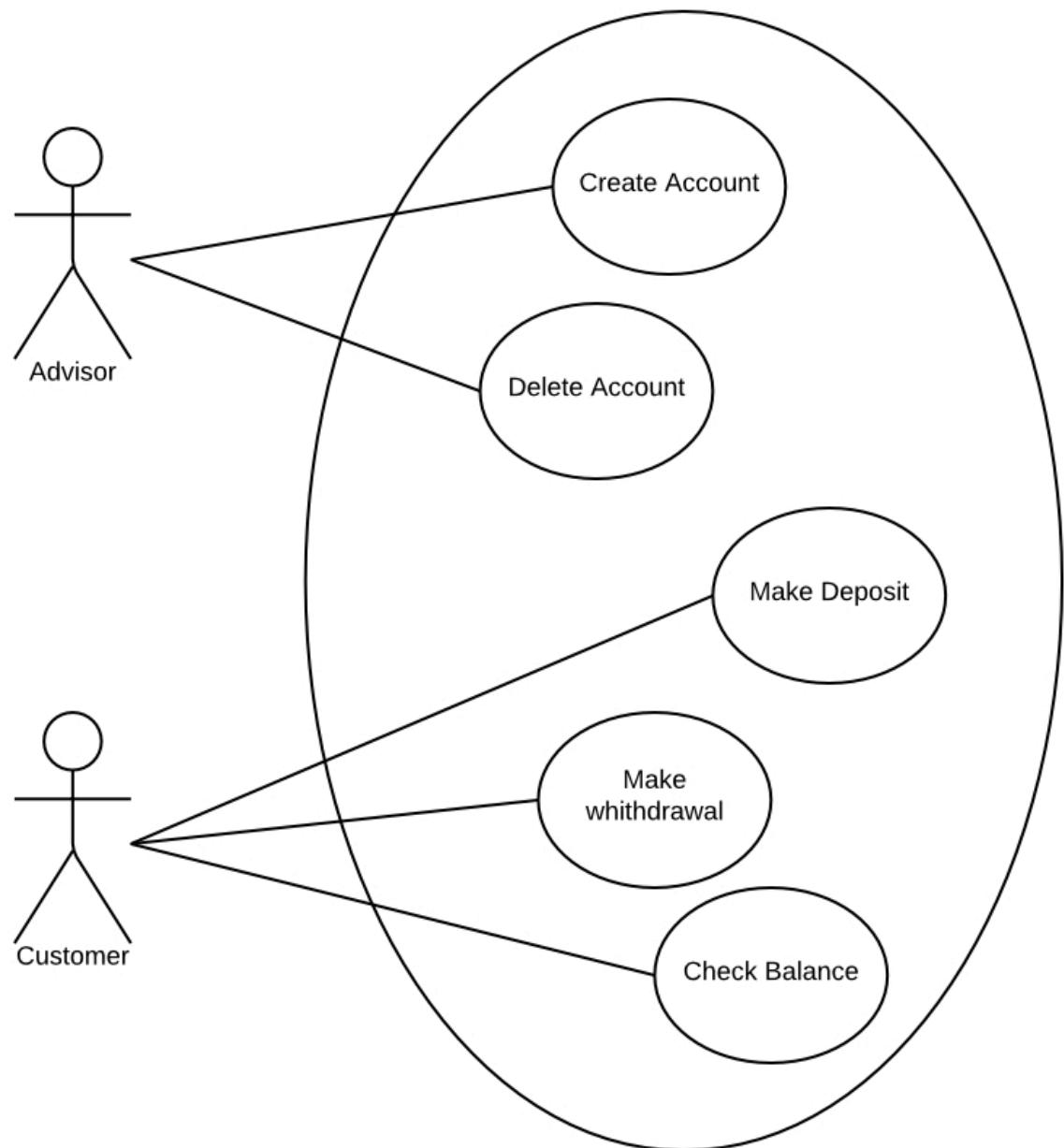
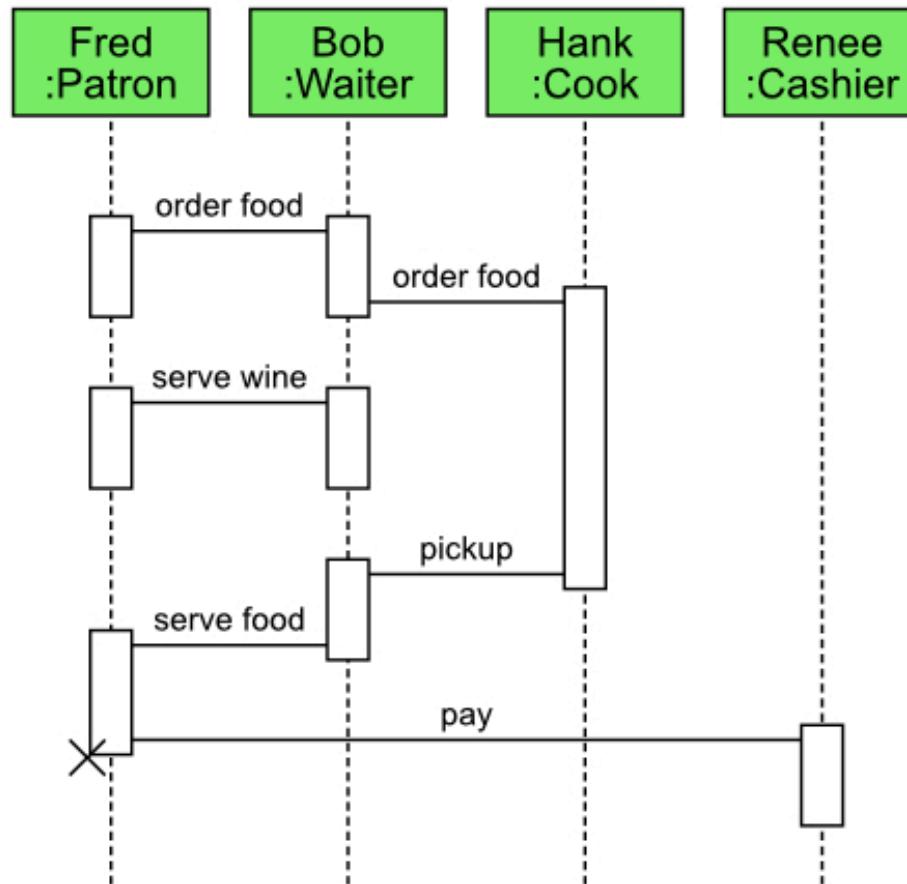


DIAGRAMME SÉQUENCE



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__`...
- ▶ Pour voir si un objet a un comportement :
`getattr(objet, méthode, valeur par défaut)`
- ▶ Pour comparer des objets : `isinstance(objet)`

DARKO STANKOVSKI

PYTHON : LE TRI

FONCTION ET MÉTHODE DE TRI

- ▶ Les listes disposent de la méthode `sort()`
- ▶ La méthode `sort()` tri la liste sur laquelle elle est appelée
- ▶ La fonction `sorted(iterable)` retourne une liste triée de l'itérable passé en paramètre
- ▶ `sort` et `sorted` prennent 3 arguments optionnels : `cmp`, `key` et `reverse`

FONCTION ET MÉTHODE DE TRI - CRITÈRE DE TRI

- ▶ `reverse` est une valeur booléenne (défaut `False`)
- ▶ `key` spécifie une fonction attendant un argument qui retourne une valeur utilisée pour la comparaison
- ▶ `cmp` spécifie une fonction attendant deux arguments et qui retournera une valeur positive, négative ou nulle selon que le premier élément soit supérieur, inférieur ou égal au second.
- ▶ `cmp` est réputé lent, préférer l'usage de `key`

FONCTION ET MÉTHODE DE TRI - CRITÈRE DE TRI

```
>>> # people est une liste de type Person  
  
>>> people.sort(reverse=True)  
  
>>> people.sort(key=lambda p: p._last_name)  
  
>>> def compare_people(a, b):  
...     return cmp(a._last_name, b._last_name)  
...  
>>> sorted(people, cmp=compare_people)
```

DARKO STANKOVSKI

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*.

DÉCLENCHE 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__():
        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()
```

DARKO STANKOVSKI

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

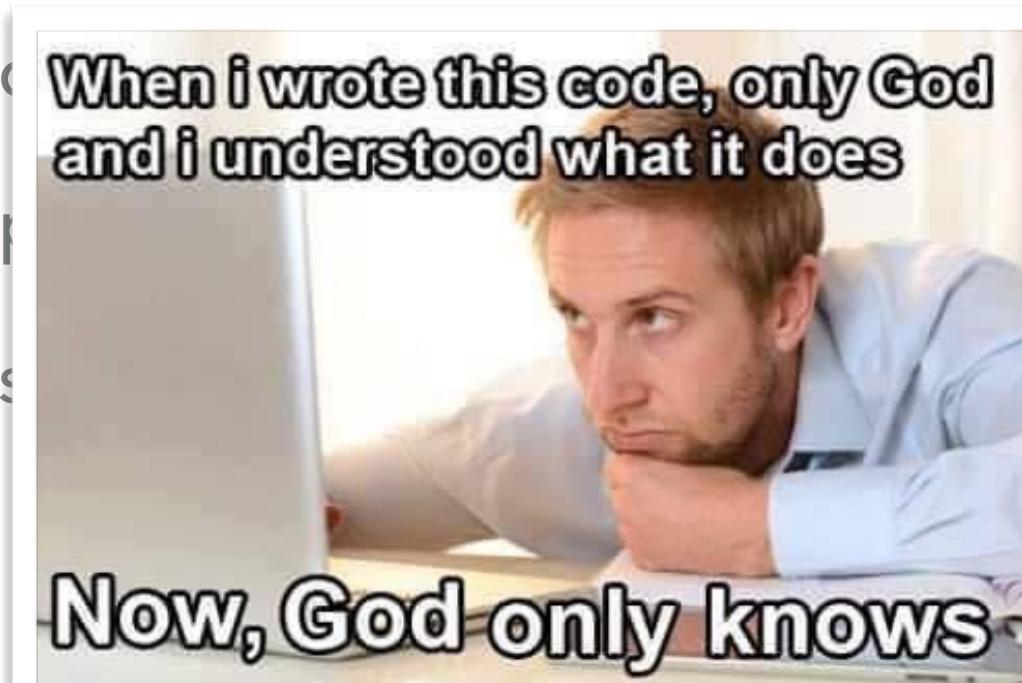
- ▶ La mise en place des tests est considérée comme une perte de temps
- ▶ Le respect des normes et métriques se heurte au délais de livraison
- ▶ Conséquence : augmentation de la dette technique

DARKO STANKOVSKI

PYTHON : DOCUMENTATION

POURQUOI DOCUMENTER ?

- ▶ Informer des autres (autres développeurs, collègues...) de la manière dont fonctionnent vos fonctions, méthodes, objets...)
- ▶ Informer les futurs utilisateurs de la manière dont utiliser vos fonctions, méthodes, objets...)
- ▶ Informer des futurs utilisateurs de la manière dont utiliser vos objets...)



DEUX OUTILS

- ▶ Les commentaires
- ▶ Les docstrings

LES COMMENTAIRES

- ▶ Un commentaire commence par un croisillon `#`
(Qui n'est pas un dièse : `#`)
- ▶ 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 docstrings 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](http://thomas-cokelaer.info/tutorials/sphinx/docstring_python.html)

LES DOCSTRINGS

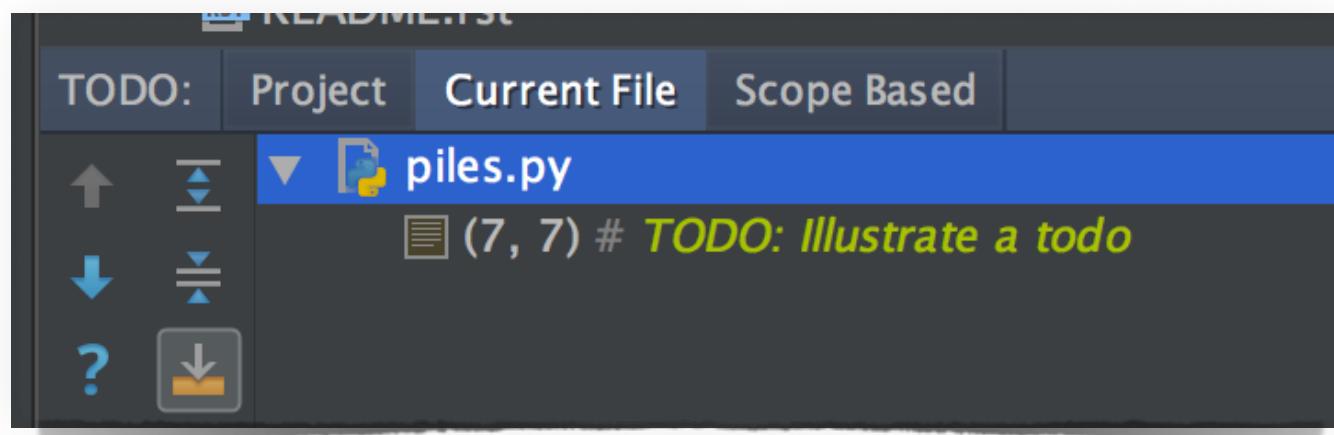
```
def add(a, b):
    """
    Adds two numbers and returns the result.

    :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 completely 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

DARKO STANKOVSKI

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/3.6/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

```
import unittest
from training.poo.bank import bank

class TestCreateAccount(unittest.TestCase):

    def testBasicAccount(self):
        account = bank.BankAccount('012345')
        self.assertEqual(100, account.balance())

    def testAccountWithBalance(self):
        account = bank.BankAccount('012345', 500)
        self.assertEqual(500, account.balance())

    def testAccountWithBalanceAsString(self):
        account = bank.BankAccount('012345', '500')
        self.assertEqual(500, account.balance())
```

UNITTEST

```
import unittest
from training.poo.bank import bank

class TestDeposit(unittest.TestCase):

    def setUp(self):
        self.account = bank.BankAccount('012345', 500)

    def testBasicDeposit(self):
        self.account.deposit(100)
        self.assertEqual(600, self.account.balance())

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

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 TestDeposit(unittest.TestCase):

    def testNegativeDeposit(self):
        with self.assertRaises(ValueError):
            self.account.deposit(-100)
```

```
class TestDeposit(unittest.TestCase):

    def testNegativeDeposit(self):
        with self.assertRaises(ValueError) as context:
            self.account.deposit(-100)

        self.assertTrue('Negative value' 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

DARKO STANKOVSKI

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'arrêt 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/3.6/library/profile.html>
- ▶ Exemple d'exécution :
`python -m cProfile -s cumtime my_script.py`

DARKO STANKOVSKI

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]: my_file = open('fichier_test.txt', 'r')  
  
In [2]: my_file  
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 my_file:  
...     text = my_file.read()  
...  
...
```

LIRE ET ÉCRIRE DANS UN FICHIER

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

LIRE ET ÉCRIRE DANS UN FICHIER

- ▶ **read** : retourne le fichier comme une chaîne de caractères
- ▶ **readline** : retourne une *ligne* comme une chaîne de caractères
- ▶ **readlines** : retourne le fichier comme une liste de chaînes 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. Offset négatif uniquement pour les fichiers binaires. Pour texte seul 0 accepté

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
- ▶ `for line in f` : itère sur le fichier
- ▶ **Attention** : on ne peut pas manipuler le curseur si on utilise l'itérateur car celui-ci fonctionne avec un buffer.

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

DARKO STANKOVSKI

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',
                           passwd='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 = "Yoda"  
age = 800  
  
cur.execute("INSERT INTO people VALUES (?, ?)", (who, age))
```

EXÉCUTER UNE REQUÊTE

```
who = "Yoda"  
age = 800  
  
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

DARKO STANKOVSKI

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 TIMDELTA

- ▶ 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 chaîne de caractères
- ▶ Voir
<https://docs.python.org/3.6/library/time.html#time.strftime>
- ▶ Formatage appliqué par strftime

FORMATER LES DATES

```
>>> import datetime  
>>> datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S.%f')  
'2015-12-16 13:30:54.995830'
```

LIRE LES DATES

```
from datetime import datetime  
  
date_object = datetime.strptime('Jun 1 2005  1:33PM', '%b %d %Y %I:%M%p')
```

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, 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)
```

DARKO STANKOVSKI

PYTHON : REGEX

QU'EST-CE QUE LES REGEX ?

- ▶ Les expressions régulières sont un puissant moyen pour rechercher ou d'isoler des expressions d'une chaîne de caractères.
- ▶ Permet de rechercher un motif
- ▶ Permet de *valider* une chaîne de caractères
- ▶ Python fournit le module [re](#)
<https://docs.python.org/2/library/re.html>

DÉFINIR UN MOTIF

- ▶ Définir une chaîne de caractères décrivant le motif
- ▶ Marqueurs pour indiquer les caractéristiques
- ▶ 'chat' permet de rechercher le motif 'chat' et le trouvera également dans 'chaton', 'achat' ou 'un chat noir'

DÉFINIR UN MOTIF : DÉBUT OU FIN DE CHAINE

- ▶ `^` précédent le motif impose de trouver le motif en début de chaine.
`'^chat'` reconnaît `'chat'`, `'chaton'` mais pas `'achat'` ni `'un chat noir'`

- ▶ `$` suivant le motif impose de trouver le motif en fin de chaine.
`'chat$'` reconnaît `'chat'`, `'achat'` mais pas `'chaton'` ni `'un chat noir'`

DÉFINIR UN MOTIF : CLASSES DE CARACTÈRES

L'usage de crochets permet de définir les caractères acceptés

- ▶ **[aeiouy]** : un caractère parmi les voyelles
- ▶ **[a-d]** : un caractère entre a et d, identique à **[abcd]**
- ▶ **[a-dA-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 : CLASSES DE CARACTÈRES

C-3PO

K-2SO

[A-Z]-[0-9][A-Z][A-Z]

R2-D2

Chopper

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 occurrence
- ▶ {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

C-3PO

[A-Z]-[0-9][A-Z]+

K-2SO

[A-Z]-[0-9][A-Z]{2}

R2-D2

[A-Z][0-9]?-

Chopper

DÉFINIR UN MOTIF : LES GROUPES

Les parenthèses permettent de définir des groupes de caractères

- ▶ `(chat)+` : est reconnu dans '**chat**' et '**tchatchat**'
- ▶ `(l[iao])\{3}` : identifie '**lilalo**' ou '**lilola**' ou...
- ▶ `(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 chaîne est précédée par un `r`
`r'\\n'`

UTILISER LE MODULE RE : SEARCH

- ▶ `re.search(pattern, seq)` recherche le pattern dans seq
- ▶ 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 chaîne

UTILISER LE MODULE RE : SEARCH

```
>>> re.search('[A-Z][0-9]?-', 'C-3PO')
<_sre.SRE_Match at 0x11099e510>
>>> re.search('[A-Z][0-9]?-', 'K-2SO')
<_sre.SRE_Match at 0x11099e578>
>>> re.search('[A-Z][0-9]?-', 'Chopper')
>>> re.search('[A-Z][0-9]?-', 'R2-D2')
<_sre.SRE_Match at 0x11099e5e0>
```

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 défini 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]{2,})')
>>> 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'
```

UTILISER LE MODULE RE : AFFICHER LE RÉSULTAT

```
>>> float_regex = re.compile(\n... ' (?P<whole>[0-9]+)\.(?P<fraction>[0-9]{2,})')\n>>> ma_chaine = 'pi vaut 3.14'\n>>> resultat = float_regex.search(ma_chaine)\n>>> resultat.group()\n'3.14'\n>>> resultat.group(0)\n'3.14'\n>>> resultat.group(1)\n'3'\n>>> resultat.group('fraction')\n'14'
```

UTILISER LE MODULE RE : AFFICHER LE RÉSULTAT

```
>>> float_regex = re.compile(\n... ' (?P<whole>[0-9]+)\.(?P<fraction>[0-9]{2,})')\n>>> ma_chaine = 'pi vaut 3.14'\n>>> resultat = float_regex.search(ma_chaine)\n>>> resultat.group()\n'3.14'\n>>> resultat.group('fraction')\n'14'\n>>> resultat.start()\n8\n>>> resultat.end()\n12\n>>> resultat.start('fraction')\n10
```

UTILISER LE MODULE RE : TROUVER TOUTES LES OCCURRENCES

- ▶ `search` ne renvoi que la première occurrence
- ▶ `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]{2,}')
>>> 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]{2,}')
```

```
>>> 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'
```

DARKO STANKOVSKI

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_around_the_function_originale():
...         print("Avant que la fonction ne s'exécute")
...         fonction_a_decorrer()
...         print("Après que la fonction se soit exécutée")
...     return wrapper_around_the_function_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)
```

DARKO STANKOVSKI

PYTHON : PRÉSENTATION FRAMEWORKS WEB

QUELQUES FRAMEWORKS

- ▶ Django
- ▶ Bottle
- ▶ Pyramid : Compétiteur de Django influencé par Zope
- ▶ Twisted : Framework Internet
- ▶ Flask, Cherrypy, web2py, Tornado...

DARKO STANKOVSKI

PYTHON : BOTTLE

BOTTLE

- ▶ Plus petit framework disponible : tient en 1 fichier
- ▶ Outil d'enseignement, apprentissage, petits projets ou code jetable
- ▶ S'installe via pip
pip install bottle
- ▶ <http://bottlepy.org/>

BOTTLE : HELLO WORLD

```
from bottle import route, run, template

@route('/hello/<name>')
def index(name):
    return template('<b>Hello {{name}}</b>!', name=name)

run(host='localhost', port=8080)
```

BOTTLE : REQUÊTES

```
from bottle import get, post, request # or route

@post('/login') # or @route('/login', method='POST')
def do_login():
    username = request.forms.get('username')
    password = request.forms.get('password')
    if check_login(username, password):
        return "<p>Your login information was correct.</p>"
    else:
        return "<p>Login failed.</p>"
```

BOTTLE : PAGES D'ERREUR

```
from bottle import error

@error(404)
def error404(error):
    return 'Nothing here, sorry'
```

BOTTLE : GESTION DE PARAMÈTRES

```
from bottle import route, request, response, template

@route('/forum')
def display_forum():
    forum_id = request.query.id
    page = request.query.page or '1'
    return template('Forum ID: {{id}} (page {{page}})',  
                  id=forum_id, page=page)
```

BOTTLE : FICHIERS STATIQUES

```
from bottle import static_file
@route('/static/<filename>')
def server_static(filename):
    return static_file(filename,
                      root='/path/to/your/static/files')
```

```
@route('/static/<filepath:path>')
def server_static(filepath):
    return static_file(filepath,
                      root='/path/to/your/static/files')
```

BOTTLE : TEMPLATES

- ▶ Sont mis en forme par la fonction `template()` ou le décorateur `view()`
- ▶ Par défaut, doivent être localisés dans le répertoire `./views/`
- ▶ Un autre répertoire peut être ajouté à la liste `bottle.TEMPLATE_PATH`

BOTTLE : TEMPLATES

```
@route('/hello')
@route('/hello/<name>')
def hello(name='World'):
    return template('hello_template', name=name)
```

```
@route('/hello')
@route('/hello/<name>')
@view('hello_template')
def hello(name='World'):
    return dict(name=name)
```

BOTTLE : TEMPLATES

```
%if name == 'World':  
    <h1>Hello {{name}}!</h1>  
    <p>This is a test.</p>  
%else:  
    <h1>Hello {{name.title()}}!</h1>  
    <p>How are you?</p>  
%end
```

BOTTLE : TEMPLATES

```
<ul>
    % for item in basket:
        <li>{{item}}</li>
    % end
</ul>
```

BOTTLE : TEMPLATES

- ▶ Les templates sont des fichiers avec extension .tpl
- ▶ Les templates sont mis en cache après compilation
- ▶ Les templates en cache ne sont pas rechargés
- ▶ Le chargement est forcé en vidant cache par instruction
bottle.TEMPLATES.clear()
- ▶ Le cache est désactivé en mode débug

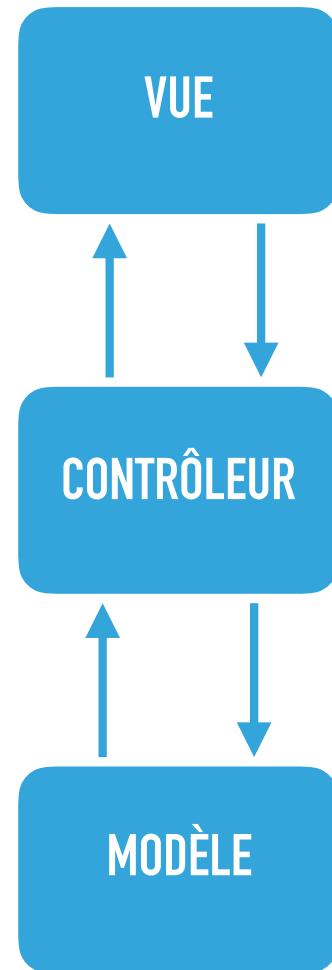
DARKO STANKOVSKI

PYTHON : DJANGO

DJANGO

- ▶ ORM (Object-Relational Mapping) et API d'accès aux données
- ▶ Langages de templates
- ▶ Mapping d'URLS
- ▶ Administration préconfigurée
- ▶ Gestionnaire de cycle de vie du projet

BASÉ SUR LE PATTERN MVC



DJANGO : LEXIQUE

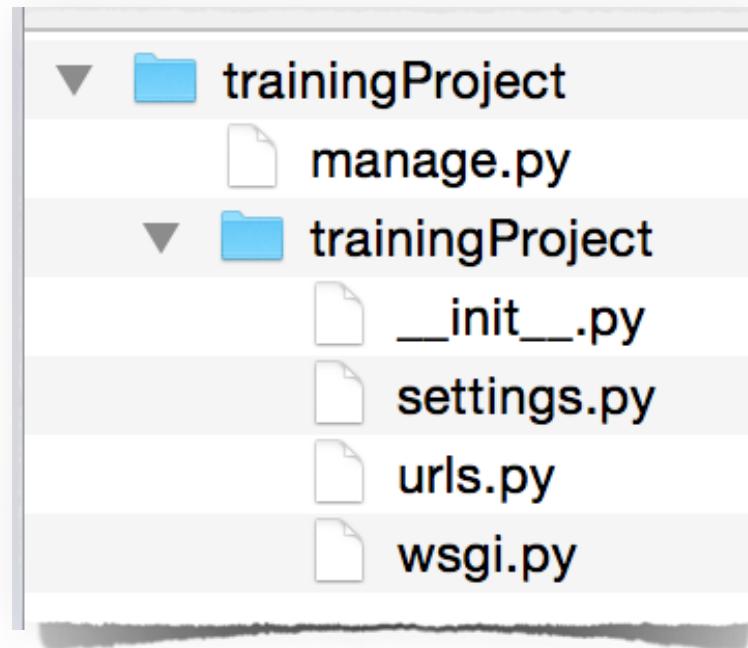
- ▶ App : Une application (web) qui fait quelque chose
- ▶ Projet : collection de configurations et apps pour une application web

Un projet peut contenir plusieurs apps, une app peut faire parti de plusieurs projets.

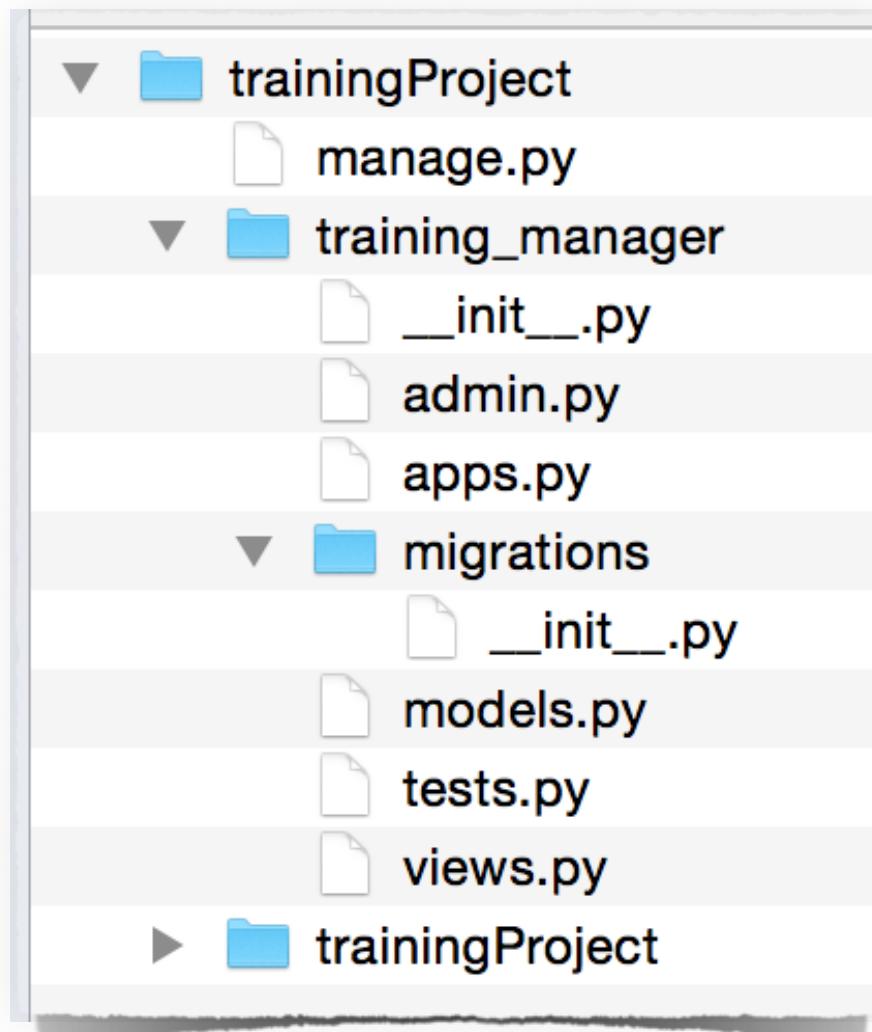
DJANGO : GESTIONNAIRE DE PROJET

- ▶ `django-admin startproject nom_projet` : crée un projet
- ▶ `python manage.py startapp mon_app` : crée une app
- ▶ `python manage.py runserver` : démarre le serveur de dev
- ▶ `python manage.py migrate` : permet de mettre à jour le modèle
- ▶ `python manage.py shell` : lance un shell interactif avec l'environnement configuré

DJANGO : STRUCTURE D'UN PROJET



DJANGO : STRUCTURE D'UNE APP



QU'EST-CE QUE LE MODÈLE POUR DJANGO ?

- ▶ Référence des données métier manipulées par Django
- ▶ Définition des champs et des comportements des données
- ▶ Facilite la persistance et le chargement
- ▶ MVC : correspond au Modèle - au contenu de la Base de Données

MODÈLE : EXEMPLE

```
from django.db import models

class Training(models.Model):
    title = models.CharField(max_length=150)
    duration = models.IntegerField()
    price = models.IntegerField()

class ClassRoom(models.Model):
    reference = models.CharField(max_length=20)
    capacity = models.IntegerField()
```

MANIPULER DES OBJETS

```
>>> t = Training(title='Python', duration=5, price=2500)
>>> t.save()
>>> t.id
1
>>> t.name
'Python'
>>> t.name = 'Swift'
>>> t.save()
```

MANIPULER DES OBJETS

```
>>> Training.objects.all()
[<Training: Swift>, <Training: Python>]
>>> Training.objects.filter(id=2)
[<Training: Python>]
>>> Training.objects.filter(name__startswith='Swi')
[<Training: Swift>]
```

DJANGO : LES VUES

- ▶ Vues définies dans views.py
- ▶ Configuration des URLs dans urls.py
- ▶ Les vues sont définies par app
- ▶ Chaque app définit sa structure d'URLs
- ▶ La configuration des URLs au niveau projet doit diriger vers les apps et déléguer la gestion d'URLs

DJANGO : LES VUES

```
from django.http import HttpResponse

def index(request):
    return HttpResponse(
        "Hello, world. You're at the training index.")
```

DJANGO : LES URLs DE L'APP

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^$', views.index, name='index'),
]
```

DJANGO : LES URLs DU PROJET

```
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
    url(r'^training/', include('training_manager.urls')),
    url(r'^admin/', admin.site.urls),
]
```

DJANGO : LES VUES

```
def detail(request, training_id):
    return HttpResponse("You're looking for training %s."
                        % training_id)

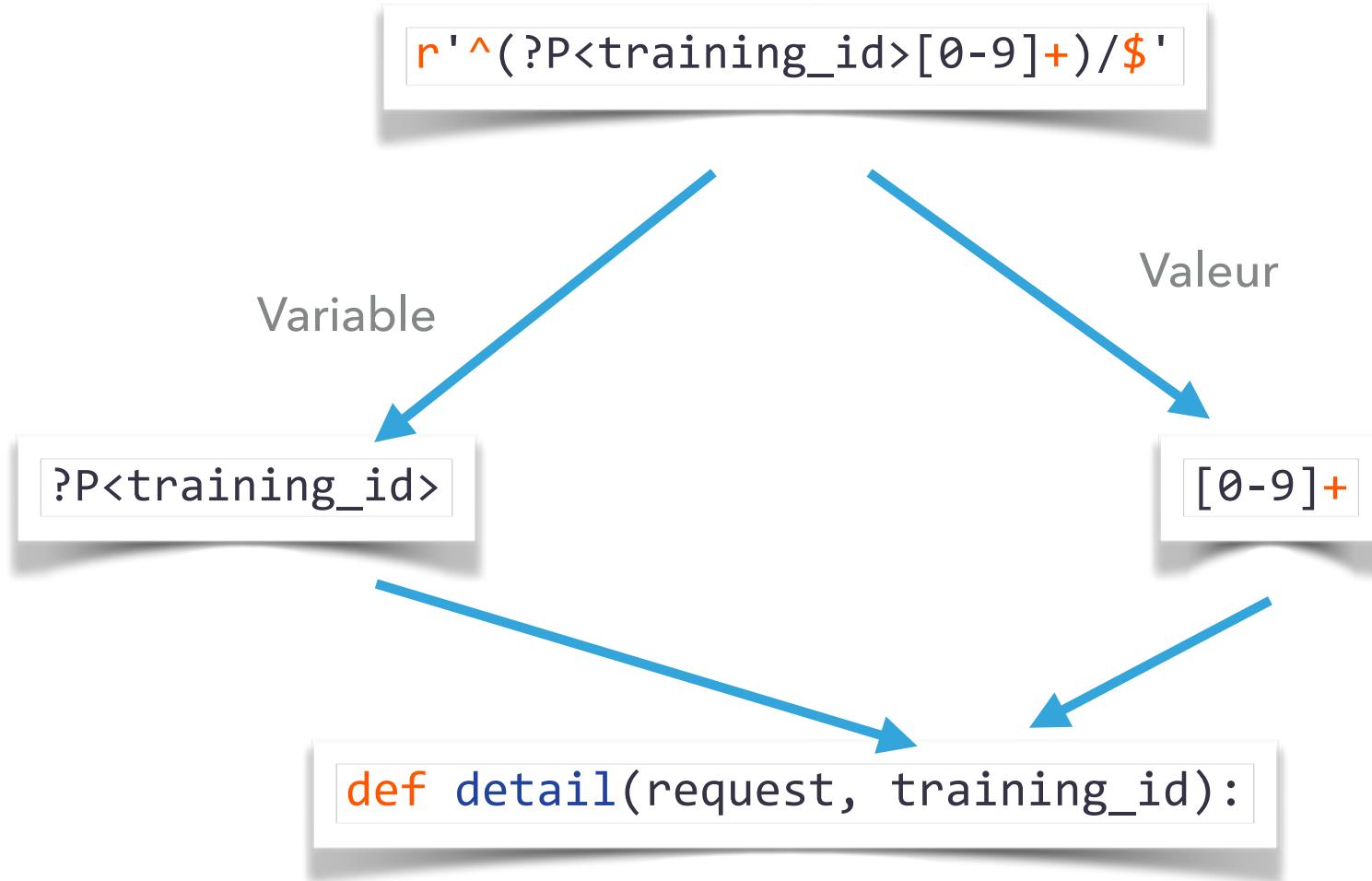
def availability(request, training_id):
    response = "You're looking for availability of training %s."
    return HttpResponse(response % training_id)
```

DJANGO : LES URLs DE L'APP

```
from django.conf.urls import url
from . import views

urlpatterns = [
    # ex: /training/
    url(r'^$', views.index, name='index'),
    # ex: /training/5/
    url(r'^(?P<training_id>[0-9]+)/$', views.detail, name='detail'),
    # ex: /training/5/availability/
    url(r'^(?P<training_id>[0-9]+)/availability/$', views.results, name='availability'),
]
```

DJANGO : LES URLs DE L'APP



DJANGO : LES TEMPLATES

- ▶ Séparent la vue du traitement
- ▶ Permettent la maintenance de l'apparence visuelle
- ▶ Par défaut, Django recherche les templates dans le répertoire
template/

DJANGO : LES TEMPLATES

```
from django.shortcuts import render

from .models import Session

def index(request):
    latest_training_list = Session('-start_date')[5]
    context = {'latest_session_list': latest_session_list}
    return render(request, 'training/index.html', context)
```

DJANGO : LES TEMPLATES

```
{% if latest_sassion_list %}  
    <ul>  
        {% for session in latest_session_list %}  
            <li><a href="/training/{{ session.subject.id }}/">  
                {{ session.subject.title }}</a></li>  
        {% endfor %}  
    </ul>  
{% else %}  
    <p>No sessions are available.</p>  
{% endif %}
```

template/training/index.html

ADMINISTRATION : PHILOSOPHIE

- ▶ Automatiser toute ce composant *sans valeur ajoutée*
- ▶ Cohérence des données définie par les modèles
- ▶ Administration n'est pas une partie publique
- ▶ Séparer contenu de la présentation

ADMINISTRATION

- ▶ Django génère automatiquement une interface d'administration
- ▶ Chaque app configure ce qui est géré par l'interface d'administration

ADMINISTRATION

```
from django.contrib import admin  
  
from .models import Training, Session  
  
admin.site.register(Training)  
admin.site.register(Session)
```

ADMINISTRATION

Django administration

Site administration

AUTHENTICATION AND AUTHORIZATION

Groups

 Add  Change

Users

 Add  Change

TRAINING_MANAGER

Sessions

 Add  Change

Trainings

 Add  Change

RAPPEL DU MODÈLE

```
class Training(models.Model):
    title = models.CharField(max_length=150)
    duration = models.IntegerField('Duration (days)')
    price = models.IntegerField('Price (euro)')
    is_active = models.BooleanField(default=True)

    def __unicode__(self):
        return self.title

class Session(models.Model):
    subject = models.ForeignKey(Training, on_delete=models.CASCADE,
                               limit_choices_to={'is_active': True})
    start_date = models.DateField('Training start date',
                                  help_text='There is no validation of the day '
                                            'in week when the training starts.')

    def __unicode__(self):
        return "%s (%s)" % (self.subject.title, self.start_date)
```

ADMINISTRATION

Django administration

Home › Training_Manager › Sessions › Add session

Add session

Subject:

✓ -----
Python
Swift

Training start date:

Today |

There is no validation of the day in week when the training starts.

DARKO STANKOVSKI

PYTHON : TKINTER

PRÉSENTATION

- ▶ Bibliothèque graphique intégrée de base à Python
- ▶ Basée sur Tk
- ▶ L'avantage est la portabilité
- ▶ Attention aux changement de noms entre Python 2 et 3

HELLO WORLD

```
from tkinter import *

fenetre = Tk()

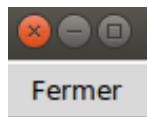
label = Label(fenetre, text="Hello World")
label.pack()

fenetre.mainloop()
```

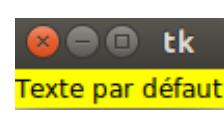


WIDGETS

Button



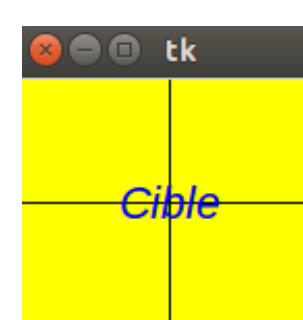
Label



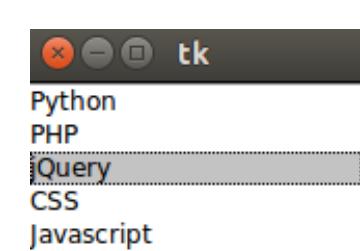
Entry



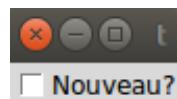
Canvas



Listbox



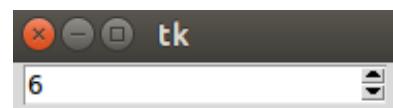
Checkbutton
Radiobutton



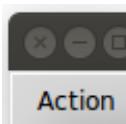
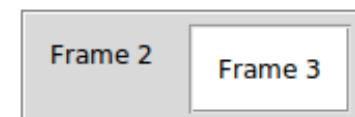
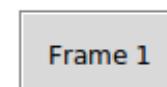
Scale



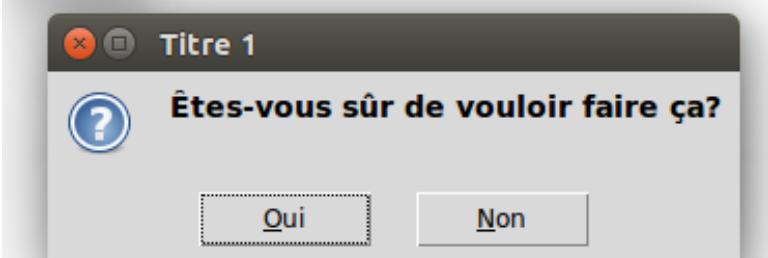
Spinbox



Frame



messagebox



AJOUTER UN WIDGET

- ▶ Créer puis positionner le widget
- ▶ Le premier paramètre est le conteneur
- ▶ Des attributs permettent de paramétrer le widget

```
un_widget = UnWidget(widget_parent, un_parametre='une_valeur')
```

PARAMÈTRES COMMUNS

text	Texte affiché par la widget
foreground (fg)	Couleur du texte du widget.
activeforeground	Couleur du texte lorsque le curseur est sur le widget.
background (bg)	Couleur de fond.
activebackground	Couleur de fond lorsque le curseur est sur le widget
padx	Marge horizontale entre les bords du widget et son contenu.
pady	Marge verticale entre les bords du widget et son contenu.
width	Largeur du widget en taille de police.
height	Hauteur du widget en taille de police.

AGENCER LES COMPOSANTS

- ▶ `pack()`
- ▶ `place()`
- ▶ `grid()`

AGENCER LES COMPOSANTS – PACK

- ▶ La méthode `pack()` accepte un paramètre `side`
- ▶ `Side` peut avoir les valeurs `TOP` (défaut), `LEFT`, `RIGHT` et `BOTTOM`
- ▶ `Pack` divise le conteneur en deux zones et place le widget du coté déclaré par `side`

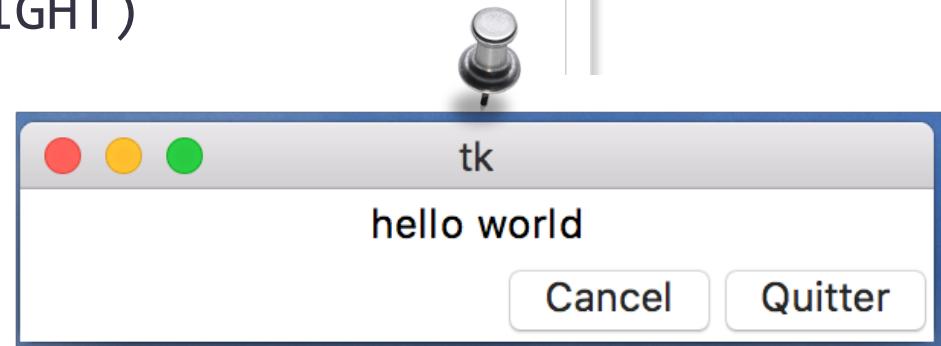
AGENCER LES COMPOSANTS - PACK

```
from tkinter import *
fen = Tk()
fen.minsize(300, 40)
label = Label(fen, text='hello world')
label.pack()

button_quit = Button(fen, text='Quitter')
button_quit.pack(side=RIGHT)

button_cancel = Button(fen, text='Cancel')
button_cancel.pack(side=RIGHT)

fen.mainloop()
```

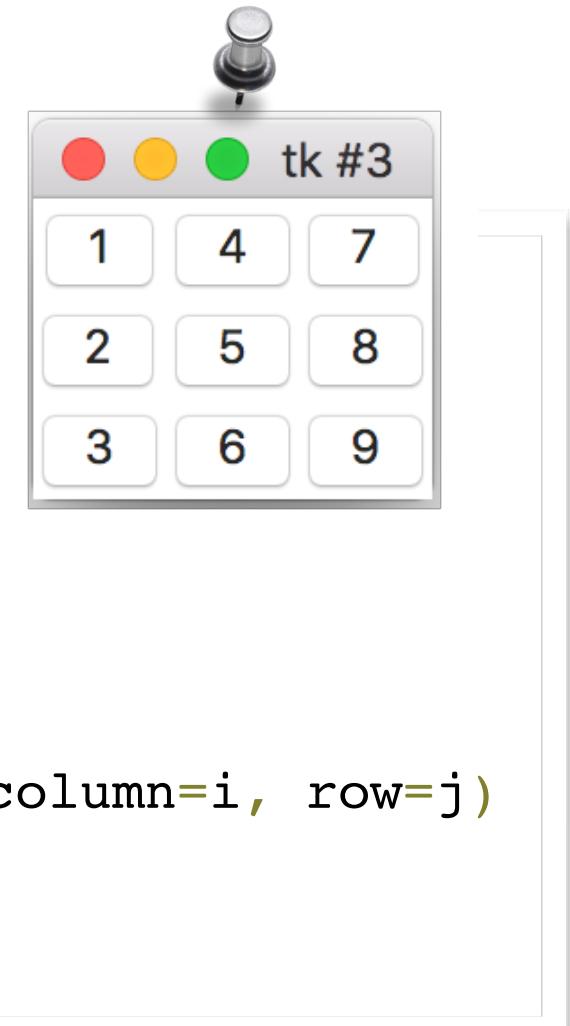


AGENCER LES COMPOSANTS - GRID

- ▶ La méthode `grid()` accepte les paramètres `column` et `row`
- ▶ Les indices de `column` et `row` commencent à 0 (haut gauche)
- ▶ On n'est pas obligé de placer les composants dans l'ordre

AGENCER LES COMPOSANTS - GRID

```
from tkinter import *
fenetre = Tk()
value = 1
for i in range(3):
    for j in range(3):
        Button(fenetre, text=value).grid(column=i, row=j)
        value += 1
fenetre.mainloop()
```



INTERACTIONS : LES BOUTONS

```
from tkinter import *

fenetre = Tk()

bouton=Button(fenetre, text="Fermer", command=fenetre.quit)
bouton.pack()

fenetre.mainloop()
```

INTERACTIONS : LES BOUTONS

```
from tkinter import *

fenetre = Tk()

def log_text():
    print("hello world")

Button(fenetre, text='Log', command=log_text).pack()

fenetre.mainloop()
```

INTERACTIONS : LES SAISIES

```
from tkinter import *

fenetre = Tk()

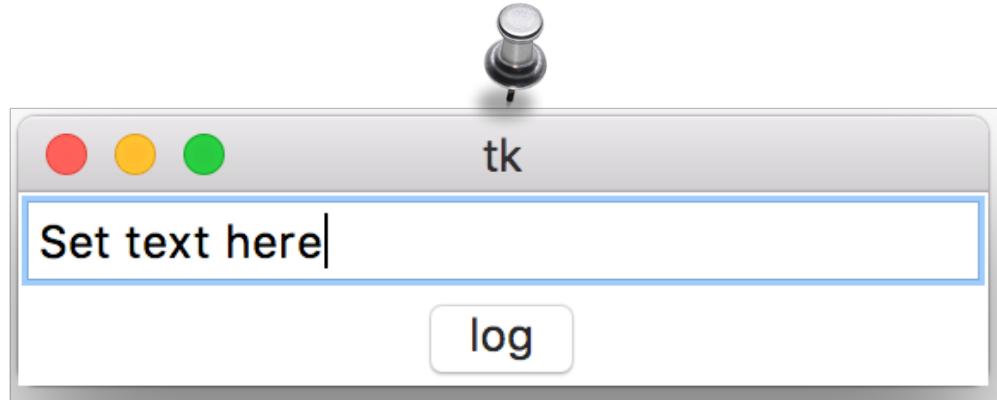
value = StringVar()
value.set('Set text here')

Entry(fenetre, textvariable=value, width=30).pack(side=TOP)

def log_value():
    print(value.get())

Button(fenetre, text='log', command=log_value).pack()

fenetre.mainloop()
```



INTERACTIONS : LES SAISIES

```
from tkinter import *

fenetre = Tk()

checkValue = IntVar()
bouton = Checkbutton(fenetre, text='un label',
                      variable=checkValue,
                      onvalue=5, offvalue=0)
bouton.pack()

def get_value():
    if checkValue.get():
        print('value is %d' % checkValue.get())
    else:
        print('not Checked')

Button(fenetre, text='log', command=get_value).pack()

fenetre.mainloop()
```



INTERACTIONS : LES SAISIES - LISTBOX

```
fenetre = Tk()  
  
choices = Variable(fenetre, ('Java', 'Python', 'Swift'))  
listbox = Listbox(fenetre,  
                  listvariable=choices,  
                  selectmode='single')  
  
listbox.pack()
```



listvariable=choices,
selectmode='single')

- ▶ single
- ▶ browse
- ▶ multiple
- ▶ extended

INTERACTIONS : LES SAISIES - LISTBOX

Ajouter des éléments

```
listbox.insert(END, 'Ruby')  
  
listbox.insert(0, 'Pascal')  
  
listbox.insert(10, 'Scala')
```

Ces éléments sont ajoutés à la variable choices

INTERACTIONS : LES SAISIES - LISTBOX

Récupérer la sélection

```
for indice in listbox.curselection():
    print(choices.get()[indice])
```

ORGANISATION DU CODE

- ▶ Pour les programmes conséquents, créer ses propres composants
- ▶ Regrouper les widgets et la logique dans des classes héritant de Frame
- ▶ Implanter la logique et permettre la communication via des méthodes dédiées

ORGANISATION DU CODE

```
class UserInfo(Frame):
    def __init__(self, master=None, cnf={}, **kw):
        Frame.__init__(self, master, cnf, **kw)
        Label(self, text='nom').grid(column=0, row=0)
        Label(self, text='prenom').grid(column=0, row=1)

        self.last_name_value = StringVar()
        self.first_name_value = StringVar()

        Entry(self, textvariable=self.last_name_value, width=30) \
            .grid(column=1, row=0)
        Entry(self, textvariable=self.first_name_value, width=30) \
            .grid(column=1, row=1)

    def set_defaut_values(self):
        self.first_name_value.set('John')
        self.last_name_value.set('Doe')
```

ORGANISATION DU CODE

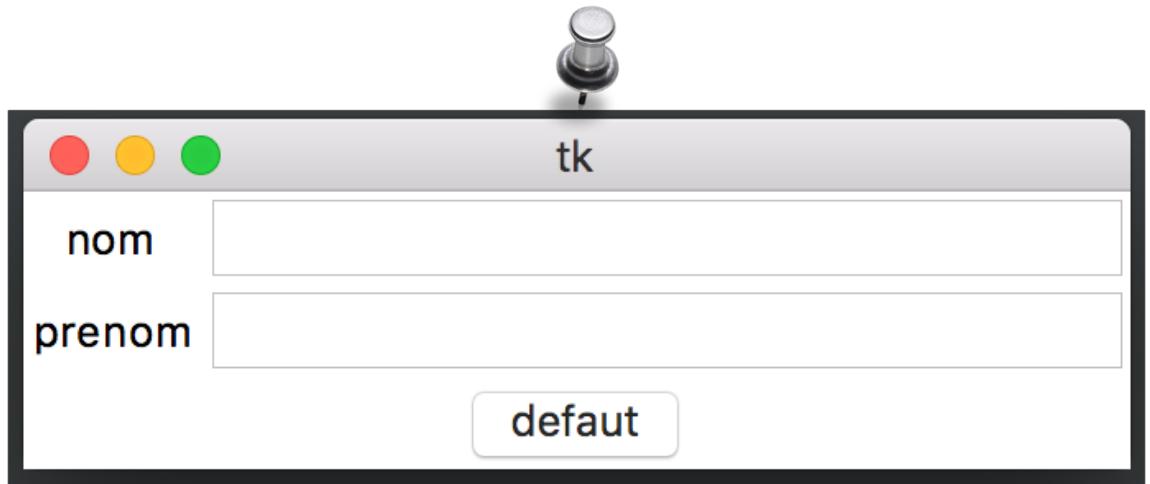
```
from tkinter import *

root = Tk()
root.title('Demo')

user_frame = UserInfo(root)
user_frame.pack()

Button(root, text='defaut',
       command=user_frame.set_defaut_values).pack()

root.mainloop()
```



POUR PLUS D'INFORMATIONS

- ▶ <https://docs.python.org/3.6/library/tkinter.html>

FORMATION PYTHON

CONCLUSION