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

Formation Python: Initiation et Approfondissement

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

Formateur: LESUEUR Yohann

Présentation de la formation

Qui suis-je? « mon parcours » « mon expérience en python»

Vos attentes pour cette formations:

- Le contexte **IT**
- Votre environnement de travail
- Vos **objectifs** pour cette formation.

DAWAN - Reproduction interdite

Objectifs: initiation

Découvrir le langage, ses possibilités, ses limites

Maîtriser la **syntaxe** de base

Structurer le code: **Fonctions**, **Modules**

Comprendre et utiliser la programmation **Orientée Objet**

Avec ces acquis vous serez en mesure de :

- Installer un environnement de développement
- Construire une bibliothèque de fonctions
- Modéliser et Implémenter des classes métiers

Objectifs: Approfondissement

Manipuler les **fichiers** et les formats de données

Manipuler les **bases de données** et les **URLs**

Visualiser les données via une **interface graphique**

Avec ces acquis vous serez en mesure de :

- Construire une application complète

Introduction PYTHON

- Historique
  - Créé en 1989 par Guido van Rossum
  - 1991 : première version publique (0.9.0)
  - 2001 : Fondation Python
  - 2005 : Guido Van Rossum rejoint Google
  - 2008 : Python 3
  - 2012 : Guido Van Rossum rejoint Dropbox
  - 2016 : Python 3.6
  - 2020 : python 2.7 n’est plus maintenu

Introduction PYTHON

- Qui utilise PYTHON ?

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

  - Google (Google spider, search engine, Google app engine),YouTube (pipeline et front)
  - Instagram, Pinterest
  - Red Hat (Anaconda)
  - IBM
  - NASA
  - National Weather Service
  - Walt Disney Feature Animation

DAWAN - Reproduction interdite

Introduction PYTHON

- Caractéristiques du langage
  - Open Source
  - Langage interprété != compilé
  - Langage Multi-Plateforme : Linux, Windows, Mac OS
  - Langage Multi-Paradigme : procédural, objet, fonctionnel...
  - Langage de Programmation Haut Niveau

DAWAN - Reproduction interdite

Introduction PYTHON

- 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-go-running-python.html

Introduction PYTHON

- Langage de Haut Niveau
  - Gestion automatique de la mémoire
  - Langage fortement typé
  - Typage dynamique

DAWAN - Reproduction interdite

Introduction PYTHON

- Python2 ou Python3 ?

  - Python 2.7 n’est plus à utiliser car le support n’est plus assuré depuis janvier 2020
  - Subsistent encore des dépendances en Python 2.x

    Néanmoins

  - Depuis 2010 (2.7), Python 2 ne reçoit plus d'évolution
  - Le support Python 2 prend fin en 2020

    => Tout Nouveau projet doit reposer sur Python 3

DAWAN - Reproduction interdite

Installation PYTHON

- L'interpréteur
  - Linux : Python présent, version dépend de la distribution
  - Os X : Python 2.7 installé par défaut
  - URL d'Installation :
    https://www.python.org/downloads/
  - Installation Linux : privilégier le gestionnaires de paquets

DAWAN - Reproduction interdite

Installation PYTHON

- Installation linux via APT

```
$ sudo apt update

$ sudo apt install python3

$ python3 --version
```

DAWAN - Reproduction interdite

Installation PYTHON

- Installation linux via YUM

```
$ sudo yum makecache

$ sudo yum install yum-utils

$ sudo yum install https://centos7.iuscommunity.org/ius-release.rpm

$ sudo yum makecache

$ sudo yum install python36u

$ python3.6 --version
```

Installation PYTHON

- Installation Windows

  - Linux : télécharger installer x86 ou 64
  - Exécuter l'installeur avec les options par défaut
  - Sous Powershell en administrateur :

```
> $env:Path += ";C:\Users\[user]\Appdata\Local\Programs\Python\
Python3[v]"

> python3[v]
>>>
```

Installation PYTHON

- HELLO WORLDS !

  - Interpréteur Python

```
matthieu@matthieu:~$ python3
Python 3.7.3 (default, Apr  3 2019, 05:39:12)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("HELLO WORLD")
HELLO WORLD
>>>
```

  - Exécuter un fichier de code python

    test_install.py

```
#!/bin/python3
"""test_install.py"""
print("HELLO WORLD")
```

```
matthieu@matthieu:~$ python3 test_install.py
HELLO WORLD
matthieu@matthieu:~$
```

Installation PYTHON

- Gestionnaire de paquets PIP

  - Déjà Installé dans certains cas (≥ 2.7.9 ou ≥ 3.4)
  - URL d'Installation :
    https://pip.pypa.io/en/stable/installing/
  - Informations :
    https://pypi.python.org/pypi/pip
  - Sous Windows :

```
> $env:Path += ";C:\Users\[user]\Appdata\Local\Program\Python\Python3[v]\Scripts\"
```

Installation PYTHON

- Windows : persistence de la variable path

  - Rendre la configuration du path permanente

  - Sous powershell en administrateur :

```
> [Environment]::SetEnvironmentVariable
    ("Path", $env:Path + "C:\Users\[user]\AppData\Local\Programs\Python\
Python3[v], "Machine")
> [Environment]::SetEnvironmentVariable
    ("Path", $env:Path + "C:\Users\[user]\AppData\Local\Programs\Python\
Python3[v]\Scripts, "Machine")
```

Installation PYTHON

- Commandes PIP

```
$ sudo pip install [package]

$ sudo pip install --user [package]

$ pip install [package] –upgrade

$ pip uninstall [package]

$ sudo pip install -U pip
```

DAWAN - Reproduction interdite

Installation PYTHON

- Ipython : interpréteur évolué
  - Intérêt: introspection++, complétion++, accès shell, intégration outils graphiques
  - Installation : pip install ipython
  - Informations :
    http://ipython.org/
  - Documentation:
    http://ipython.readthedocs.io/

Installation PYTHON

- Virtualenv : environnement virtuel
  - Intérêt: contrôle et isolation des versions de l'interpréteur et des dépendances
  - Installation : pip install virtualenv
  - Définir un répertoire de travail *workdir*
  - Activation / Désactivation :

```
$ python3 -m venv /path/to/workdir

$ source /path/to/workdir/bin/activate

> \path\to\workdir\Scripts\activate.bat

[>|$] deactivate
```

Installation PYTHON

- Environnement de développement
  - Encodage requis en UTF8
  - IDLE : fourni avec installation standard, codé en Python avec tkinter
  - Visual Studio Code
    https://code.visualstudio.com/
  - PyCharm (existe en Commercial ou Community)
    https://www.jetbrains.com/pycharm/download/
  - Jupyter Notebook : éditeur en ligne de la stack Anaconda (Analyse de Données)
    https://jupyter.org/install

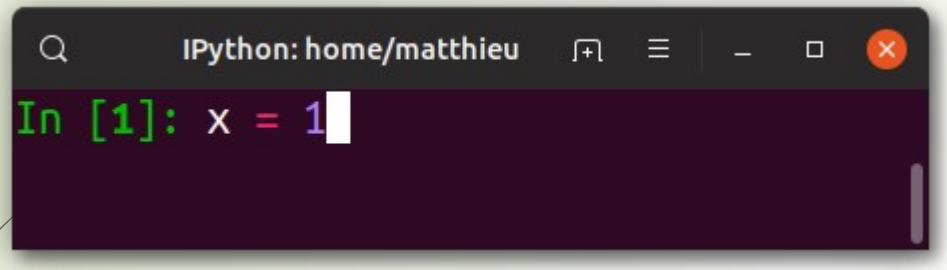
Installation PYTHON

- Portabilité python2 => python3
  - Utiliser le package future
  - http://python-future.org/

```
$ sudo pip install future
```

Syntaxe PYTHON

- Variables et Données



- Une variable référence une donnée, i.e contient une référence vers une valeur
- Le type de la variable est déterminé dynamiquement (pas de déclaration)
- Une variable est détruite lorsqu'elle n'est plus accessible (notion de portée)
- Règle de nommage : [a-zA-Z_][a-zA-Z0-9_]*

Syntaxe PYTHON

- Conventions de codage : PEP8

https://www.python.org/dev/peps/pep-0008/

- Lettres seules, en minuscule : pour les variables de type compteur, indices
- Lettres minuscules + « _ » : pour les modules, variables, fonctions et méthodes
- Lettres capitales + « _ » : pour les (pseudo) constantes
- CamelCase : Noms de Classe (voir Python orienté objet)
- **Ne jamais créer une variable avec le nom d'un mot clé python !!**

Syntaxe PYTHON

- Exemples de variables

IPython: home/matthieu

```
In [1]: answer = 42

In [2]: PI = 3.14

In [3]: hello = "HELLO World!"

In [4]: stuff = [42, "Hello", PI]

In [5]: days_of_week = ('Lundi', 'Mardi', 'Mercredi')

In [6]: nothing = None

In [7]: type(PI)
Out[7]: float
```

Syntaxe PYTHON

- Initialisation et Affectations

IPython: home/matthieu

```
In [1]: var = 42

In [2]: var_1 = var_2 = 42 #WARNING!!!

In [3]: var_1, var_2 = "Bonjour", 42
```

Syntaxe PYTHON

- Introspection

- Afficher une variable sur la sortie : print(x)
- Renseigner une variable depuis l'entrée (clavier) : x = input('message')
- Afficher l'aide sur un objet : help(x)
- Retourner le type d'une variable : type(x)
- Retourner les méthodes implicites d'une variable : dir(x)

Syntaxe PYTHON

- Les types numériques

- 3 types numériques : int (entiers), float (« réels ») et complex
- Les booléens : True et False sont des sous-types des entiers int
- On peut imposer le type (« caster ») par les fonctions int(), float() et complex()
- Représentation binaire : 0b01011111
- Représentation octale: 0o755
- Représentation hexadécimale : 0x5f
- Représentation exponentielle : 3.14e-5

DAWAN - Reproduction interdite

Syntaxe PYTHON

- Les types numériques : exemples

IPython: home/matthieu

```
In [1]: var_float = 3.14

In [2]: var_cplx = 42 + 2j

In [3]: var_cplx = complex(42, 2)

In [4]: int(var_float)
Out[4]: 3

In [5]: var_cplx.real
Out[5]: 42.0

In [6]: var_cplx.imag
Out[6]: 2.0
```

Syntaxe PYTHON

- Opérateurs sur les types numériques

x + y	Addition
x – y	Soustraction
x * y	Produit
x / y	Division
x // y	Division Entière
x % y	Modulo
-x	Opposé
+x	
x**y	Exposant

DAWAN - Reproduction interdite

Syntaxe PYTHON

- Opérateurs logiques et comparaisons

x or y	OU (Union)
x and y	ET (Intersection)
x < y	Strictement inférieur à
x > y	Strictement supérieur à
x ≤ y	Inférieur ou égal à
x ≥ y	Supérieur ou égal à
x == y	Egal à
x != y	Différent de
x is y	deux références sur le même objet
x is not y	ne référencent pas le même objet

DAWAN - Reproduction interdite

Syntaxe PYTHON

- Opérateurs binaires sur les entiers

x | y	OU
x ^ y	OU Exclusif
x & y	ET
x << n	Décalage à gauche
x >> n	Décalage à droite
~x	NON

Syntaxe PYTHON

- Priorités des opérateurs

- Règles de priorités mathématiques (** → * → +...)
- Les parenthèses ont la plus grande priorité
- Op. numériques > binaires > logiques > comparaison

https://fr.wikibooks.org/wiki/Programmation_Python/Opérateurs

Syntaxe PYTHON

- Les Séquences

- Types de séquence : string (chaînes de caractères), list, tuple (t-uplets), dict (dictionnaires)
- Chaînes de caractères et tuples sont des séquences immutables
- On peut « caster » des séquences avec les fonctions str(), list(), tuple() et dict()

=> ne peuvent être modifiés, i.e être à gauche d'une affectation

Syntaxe PYTHON

- Les Séquences : exemples

IPython: home/matthieu

```
In [1]: var_string = "Character's string"

In [2]: var_string = str(42)

In [3]: var_list = ["long", "characters", "string"]

In [4]: var_tuple = ("long", "immutable", "string")

In [5]: list(var_tuple)
Out[5]: ['long', 'immutable', 'string']

In [6]: tuple(var_list)
Out[6]: ('long', 'characters', 'string')
```

Syntaxe PYTHON

- Opérateurs communs sur string, list et tuple

x in s	True si s contient x, False sinon
x not in s	True si s ne contient pas x, False sinon
s + t	Concaténation
s * t	Répétition
s[i]	Elément à l'indice i
len(s)	Taille de la séquence
min(s)	Plus petit élément de la séquence
max(s)	Plus grand élément de la séquence
s.index(x)	Index de x dans s
s.count(x)	Nombre d'occurences de x dans s

DAWAN - Reproduction interdite

Syntaxe PYTHON

- Le slicing : générer des sous séquences

IPython: home/matthieu

```
In [1]: seq = "Hello World!"

In [2]: seq[0:5], seq[:5]
Out[2]: ('Hello', 'Hello')

In [3]: seq[6:12], seq[6:], seq[-6:]
Out[3]: ('World!', 'World!', 'World!')

In [4]: x = [1,2,3]

In [5]: y, z = x, x[:]

In [6]: x == y, x == z
Out[6]: (True, True)

In [7]: y is x, z is x
Out[7]: (True, False)
```

DAWAN - Reproduction interdite

Syntaxe PYTHON

- Les Chaines de caractères (string)

- Simples quotes (‘...’) ou double quotes (‘’...’’) : sur une ligne
- Écrire une longue chaîne sur plusieurs lignes : (‘….\
- Triple-double-quotes (‘’ ’’ ’’...’’ ’’ ’’) : sur plusieurs lignes avec sauts de ligne
- Sont encodées en unicode : # coding : utf8 => auto en python3

Syntaxe PYTHON

- Opérations spécifiques sur string

IPython: home/matthieu

```
In [1]: my_str = "Hello"

In [2]: my_str2 = "World"

In [3]: my_str.capitalize()
Out[3]: 'Hello'

In [4]: my_str2.upper()
Out[4]: 'WORLD'

In [5]: my_str.split("e")
Out[5]: ['H', 'llo']

In [6]: my_str.isnumeric()
Out[6]: False

In [7]:
```

- https://www.quackit.com/python/reference/python_3_string_methods.cfm

Syntaxe PYTHON

- Affichage formaté

```
IPython: home/matthieu

In [1]: "Résultat: {}".format("success")
Out[1]: 'Résultat: success'

In [2]: "Résultat: {}, {}".format("still", "success")
Out[2]: 'Résultat: still, success'

In [3]: "Résultat: {1}, {0}".format("success", "whatever")
Out[3]: 'Résultat: whatever, success'

In [4]: "Résultat: {:5.2f}".format(23.456)
Out[4]: 'Résultat: 23.46'

In [5]: "Résultat: {value:05d} {unit}".format(unit="kg", value=3)
Out[5]: 'Résultat: 00003 kg'
```

- https://docs.python.org/3/library/string.html#formatstrings

Syntaxe PYTHON

- Les listes

- Les listes sont des séquences ordonnées
- Les listes sont mutables (modifiables)
- Les éléments d'une liste peuvent être de types différents

```
my_list = [2, 'toto']
```

Syntaxe PYTHON

- Opérations spécifiques aux listes

my_list.append(elem)	Ajoute [elem] à la fin de la liste
my_list.count(elem)	Compte le nombre d'occurences d'[elem]
my_list.extend(list2)	Concatènation de listes
my_list.insert(index, elem)	Ajoute [elem] avant l'élément d'index [index]
my_list.pop()	Dépile et retourne le dernier élément
my_list.pop(index)	Dépile et retourne l'élément d'index [index]
my_list.remove(elem)	Supprime la première occurrence d'[elem]
my_list.reverse()	Renverse la liste
my_list.sort()	Trie la liste

DAWAN - Reproduction interdite

Syntaxe PYTHON

- Les séquences : accès à un élément

IPython: home/matthieu

```
In [1]: my_str, my_list = "Python", ["Py", "th", "on"]

In [2]: my_str[0], my_list[0]
Out[2]: ('P', 'Py')

In [3]: my_str[1], my_list[1]
Out[3]: ('y', 'th')

In [4]: my_str[-1], my_list[-1]
Out[4]: ('n', 'on')
```

Syntaxe PYTHON

- Les séquences : affectations

IPython: home/matthieu

```
In [1]: my_str, my_list = "Python", ["Py", "th", "on"]

In [2]: my_list[0] = "Jy"

In [3]: my_str[0] = "J"
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-942aabfaca27> in <module>
----> 1 my_str[0] = "J"

TypeError: 'str' object does not support item assignment

In [4]: print(my_list)
['Jy', 'th', 'on']
```

=> Mutable VS Immutable

Syntaxe PYTHON

- Les listes

- Les listes sont des séquences ordonnées
- Les listes sont mutables (modifiables)
- Les éléments d'une liste peuvent être de types différents

```
my_list = [2, 'toto']
```

Syntaxe PYTHON

- Les tuples

- Les tuples sont des séquences ordonnées
- Les listes sont immutables (non modifiables)
- Les éléments d'un tuple peuvent être de types différents

```
my_tuple = (2, 'toto')

my_single_tuple = ('Hello',)
```

Syntaxe PYTHON

- Les dictionnaires

- Les dictionnaires sont des séquences non ordonnées
- Les dictionnaires sont mutables (modifiables)
- Les éléments sont des paires « clé : valeur » avec des clés uniques
- Accès aux valeurs par les clés

```
my_dict = {‘toto’ : 2, ‘tata’ : ‘non’}
```

Syntaxe PYTHON

- Maniement des dictionnaires

IPython: home/matthieu

```
In [1]: dict1, dict2 = dict(one=1, two=2) , dict([['one', 1], ['two', 2]])

In [2]: dict1['one']
Out[2]: 1

In [3]: dict2['two']
Out[3]: 2

In [4]: dict1['three'] = 3

In [5]: dict2[5] = 'five'

In [6]: print(dict1, dict2)
{'one': 1, 'two': 2, 'three': 3} {'one': 1, 'two': 2, 5: 'five'}
```

Syntaxe PYTHON

- Opérations spécifiques aux dictionnaires

my_dict.clear()	Vide le dictionnaire
my_dict.copy()	Copie du dictionnaire sans référence
my_dict.get(clef [, defaut])	Accès à la valeur de clef ou défaut
my_dict.items()	Retourne une liste de tuple (clé, valeur)
list(my_dict)	Retourne la liste des clés
list(my_dict.keys())	Retourne la liste des clefs
list(my_dict.values())	Retourne la liste des valeurs
my_dict.pop(clef)	Supprime la paire et retourne la valeur

Structures de contrôle

- Principe

- Structurer l’exécution du code selon :
- En-têtes terminés par ‘ : ‘
- Blocs de code exécutés de manière conditionnelle ou en itération (boucle)

=> En Python, les blocs sont définis par l'indentation (2 ou 4 espaces)

Structures de contrôle

- IF : structures conditionnelles

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

=> utilisation des opérateurs logiques et de comparaison

=> *condition* est évaluée comme booléen

```
if x ≥ 2 and y != x :
    bloc de code……
```

Structures de contrôle

- IF : Évaluations booléennes

Type	FAUX
bool	False
int	0
float	0.0
string	‘’
tuple	()
list	[]
dict	{}
None type	None

Structures de contrôle

- IF : exemple complet

```
#! /bin/python3

x = input("entre un nombre")
if not x.isnumeric():
  print("x n'est pas un entier !")
elif int(x) < 0:
  print("x est négatif")
else not int(x):
  print("x vaut 0 !!!")
```

```
status = "retard" if delay < 10 else "annulé"  # opérateur ternaire
```

Structures de contrôle

- La boucle FOR

- En python, la boucle FOR parcourt des séquences
- En-tête : **for** *element* **in** *sequence*:

```
1  #!/bin/python3
2
3  trainings = ['java', 'python', 'swift']
4  for t in trainings:
5    print(t, len(t))
6
7  trainees = {"java": "Bob", "python": "James", "swift": "Henry"}
8  for lang, name in trainees.items():
9    print("{} : {}".format(lang.capitalize(), name))
```

Structures de contrôle

- Boucle FOR : la fonction range()

  - Génère une séquence d'entiers sur lesquels on va boucler
  - **range**(*début_compris*, *fin_non_compris*, *pas*)
  - Dans une boucle : for i in range(…) :

```
IPython: home/matthieu

In [1]: r, r2, r3 = range(10), range(5, 11), range(0, 10, 2)

In [2]: list(r)
Out[2]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [3]: list(r2)
Out[3]: [5, 6, 7, 8, 9, 10]

In [4]: list(r3)
Out[4]: [0, 2, 4, 6, 8]
```

Structures de contrôle

- La boucle WHILE

- En python, la boucle WHILE s'exécute tant qu'une condition est vraie
- En-tête : **while *condition* :**

```
1  #!/bin/python3
2
3  a, b = 0, 1
4  while b < 10:
5      print(b)
6      a, b = b, a+b
```

Structures de contrôle

- Casser la boucle : break, continue, else

- **break :** interrompt l'exécution de la boucle et la quitte
- **continue :** interrompt l'exécution de la boucle et passe à l'itération suivante
- **else :** s'exécute à la fin de la boucle, sauf si celle ci est interrompue par break
- **pass :** instruction tampon, bloc par défaut, ne fait strictement rien

Structures de contrôle

- Les listes en intention

  - Ou encore « Comprehension lists », définissent la transformation d'une liste
  - En logique, l'intension d'un concept est sa définition

IPython: home/matthieu

```
In [1]: sequence = ["a", "b", "c"]

In [2]: new_sequence = []

In [3]: for element in sequence:
   ...:     new_sequence.append(element.upper())
   ...:

In [4]: comprehension_sequence = [element.upper() for element in sequence]

In [5]: print(new_sequence, comprehension_sequence)
['A', 'B', 'C'] ['A', 'B', 'C']
```

Structures de contrôle

- Les listes en intension : condition

➤ Structure générale

[transformation for élément in collection if condition]

IPython: home/matthieu

```
In [1]: squares = []

In [2]: for num in range(10):
   ...:     if num % 2 == 0:
   ...:         squares.append(num**2)
   ...:

In [3]: comp_squares = [num**2 for num in range(10) if num % 2 == 0]

In [4]: print(squares, comp_squares)
[0, 4, 16, 36, 64] [0, 4, 16, 36, 64]
```

Fonctions Python

- Déclaration

- Définissent des blocs de code qu'on peut paramétrer et appeler
- Retournent une valeur précédée de return ou None sinon

- Structure générale :

```
def nom_fonction([param1], [param2=default], [args*], [kwargs**]) : #signature
    bloc…
    [ return ]

    nom_fonction(param1, param2, […], {…}) # appel
```

Fonctions Python

- Premier exemple

  - Fonction factorielle

```python
#!/bin/python3

def factorielle(n):
    rslt = 1
    for i in range(1, n+1):
        rslt *= i
    return rslt
```

Fonctions Python

- Paramètres et retour
  - Ne sont pas typés

```
In [1]: def fois3(x):
   ...:     return x * 3
   ...:

In [2]: fois3(9)
Out[2]: 27

In [3]: fois3("quoi ?")
Out[3]: 'quoi ?quoi ?quoi ?'
```

  - Valeurs de retour multiples par les tuples

```
In [1]: def minimaxi(sequence):
   ...:     return min(sequence), max(sequence)
   ...:

In [2]: minimaxi([1, 2, 3])
Out[2]: (1, 3)
```

DAWAN - Reproduction interdite

Fonctions Python

- Paramètres optionnels – Valeurs par défaut

- Un paramètre peut être optionnel s'il lui est affecté une valeur dans la signature
- On peut alors omettre ce paramètre à l'appel de la fonction
- Les paramètres optionnels doivent être placé en dernier dans la signature

IPython: home/matthieu

```
In [1]: def create_account(id, name, mode="user", priority=10):
   ...:     return id, name, mode, priority
   ...:

In [2]: create_account(1, "Bob")
Out[2]: (1, 'Bob', 'user', 10)

In [3]: create_account(2, "Bill", "admin")
Out[3]: (2, 'Bill', 'admin', 10)

In [4]: create_account(3, "Bernard", priority=1)
Out[4]: (3, 'Bernard', 'user', 1)

In [5]: create_account(priority=2, id=4, name="gruut")
Out[5]: (4, 'gruut', 'user', 2)
```

Fonctions Python

- « variadics » : *args

  - Dans la signature : permet de définir des paramètres optionnels non nommés
  - Accessibles dans le corps de la fonction comme un tuple

IPython: home/matthieu

```
In [1]: def test_var_args(farg, *args):
   ...:     print(args)
   ...:     for arg in args:
   ...:         print("oarg:", arg)
   ...:

In [2]: test_var_args(1)
()

In [3]: test_var_args(1, "two", 3)
('two', 3)
oarg: two
oarg: 3
```

Fonctions Python

- « variadics » : *args

  - Dans l'appel : permet de gérer des tuples comme arguments

IPython: home/matthieu

```
In [1]: def call_var_args(arg1, arg2, arg3):
   ...:     return arg1, arg2, arg3
   ...:

In [2]: tupl = ("my name", "is")

In [3]: call_var_args("hi", *tupl)
Out[3]: ('hi', 'my name', 'is')
```

Fonctions Python

- « variadics » : **kwargs

  - Dans la signature : permet de définir des paramètres optionnels nommés
  - Accessibles dans le corps de la fonction comme un dictionnaire

IPython: home/matthieu

```
In [1]: def test_var_kwargs(farg, **kwargs):
   ...:     print(kwargs)
   ...:     for k, w in kwargs.items():
   ...:         print(k, ":", w)
   ...:

In [2]: test_var_kwargs(1)
{}

In [3]: test_var_kwargs(1, name="joe", age=33)
{'name': 'joe', 'age': 33}
name : joe
age : 33
```

Fonctions Python

- « variadics » : **kwargs

  - Dans l'appel : permet de gérer des dictionnaires comme arguments

IPython: home/matthieu

```
In [1]: def call_var_kwargs(arg1, arg2, arg3):
   ...:     return arg1, arg2, arg3
   ...:

In [2]: dct = {"arg2": "two", "arg3": 3}

In [3]: call_var_kwargs(1, **dct)
Out[3]: (1, 'two', 3)
```

Fonctions Python

- Portée d'une variable

  - Une variable est reconnue et accessible dans le bloc dans lequel elle a été créée
  - Elle est accessible dans les bloc subalternes si elle n'y a pas été redéfinie

```
IPython: home/matthieu
In [1]: x = 10

In [2]: def do_your_thing():
   ...:     print(x)
   ...:

In [3]: do_your_thing()
10
```

```
IPython: home/matthieu
In [1]: x = 10

In [2]: def do_your_local_thing():
   ...:     x = 15
   ...:     print(x)
   ...:

In [3]: do_your_local_thing()
15

In [4]: x
Out[4]: 10
```

Fonctions Python

- Portée d'une variable

  - Une variable est modifiable dans les blocs subalternes si elle y figure précédée de **global**

```
IPython: home/matthieu

In [1]: x = 10

In [2]: def do_your_global_thing():
   ...:     global x
   ...:     x = 15
   ...:     print(x)
   ...:

In [3]: do_your_global_thing()
15

In [4]: x
Out[4]: 15
```

Fonctions Python

- Fonction comme paramètre et variable

➢ Si **« def ma_fcontion() : »** est la signature, et **« ma_fonction() »** interprète la fonction

=> « ma_fonction » est une référence à la fonction qu'on utilise comme une variable

```
IPython: home/matthieu
In [1]: def my_map(func, tab):
   ...:     rslt = []
   ...:     for i in tab:
   ...:         rslt.append(func(i))
   ...:     return rslt
   ...:

In [2]: def square(x):
   ...:     return x**2
   ...:

In [3]: my_map(square, [1, 2, 3])
Out[3]: [1, 4, 9]
```

Fonctions Python

- Fonctions lambda

  - Fonctions éphémères, sans nom, constituées d'une seule instruction, à usage unique

IPython: home/matthieu

```
In [1]: def my_map(func, tab):
   ...:     rslt = []
   ...:     for i in tab:
   ...:         rslt.append(func(i))
   ...:     return rslt
   ...:

In [2]: my_map(lambda x : x**2, [1, 2, 3])
Out[2]: [1, 4, 9]

In [3]:
```

Fonctions Python

- Documentation, introspection

➢ **Docstring :** triple-double-quotes de documentation sous la signature, accessible via help()

```
IPython: home/matthieu
In [1]: def ma_fonc():
   ...:     """Je ne sers à rien!"""
   ...:     pass
   ...:

In [2]: help(ma_fonc)
```

➢ **Introspection :** la fonction dir() affiche les objets internes des fonctions

```
IPython: home/matthieu
In [3]: dir(ma_fonc)
Out[3]:
['__annotations__',
 '__call__',
 '__class__',
 '__closure__',
 '__code__',
 '__defaults__',
```

Modules Python

- Principe

- **Un module** est un fichier contenant du code python, suffixé par **« .py »**
- On peut injecter le contenu du module dans un autre fichier grâce à **import**
- **L'espace de nom** du module importé permet de reconnaître et d'utiliser ce contenu

```python
#!/bin/python3
"""Module test_module.py"""

x = 5

def factorielle(n):
    rslt = 1
    for i in range(1, n+1):
        rslt *= i
    return rslt
```

```
IPython: Documents/fo...
In [1]: import test_module

In [2]: test_module.x
Out[2]: 5

In [3]: test_module.factorielle(3)
Out[3]: 6
```

Modules Python

- bibliothèques

- **Une bibliothèque** est un module uniquement composé de fonctions à importer
- On importe directement les fonctions avec **from** *biblio* **import** funcA, funcB, *
- On renomme les fonctions importées avec **from** *biblio* **import** funcA **as** fA

```
IPython: Documents/formation_...
In [1]: from test_module import factorielle as fact

In [2]: fact(4)
Out[2]: 24
```

```
IPython: Documents/formation_...
In [1]: from test_module import *

In [2]: x
Out[2]: 5

In [3]: factorielle(5)
Out[3]: 120
```

Modules Python

- Introspection et exécution

- **Les docstrings** et la fonction **dir** peuvent être utilisés sur un le nom d'un module
- On peut réserver du code à exécuter uniquement en appelant le module en CLI

```python
#!/bin/python3
"""Module test_module.py"""

x = 5

def factorielle(n):
    rslt = 1
    for i in range(1, n+1):
        rslt *= i
    return rslt

print(x)

if __name__ == "__main__":
    print(factorielle(x))
```

```
IPython: ...
In [1]: import test_module
5
```

```
matthieu@matthieu:~/Documents/format
ion_python$ python3 test_module.py
5
120
```

Modules Python

- Structuration en Paquets

- Les paquets sont des **répertoires** contenant des modules et un fichier **« __init__.py »**
- Les paquets peuvent se décomposer en sous paquets (sous répertoires)
- Les imports de modules dans les paquets se font sous les formes :

```
import package.module

from package.module import fonction, *

from package.subpackage.module import fonction as f

#dans le dossier subpackage
from ..module  import fonction as parent_func
```

```
package
├── __init__.py
├── module.py
└── subpackage
    ├── __init__.py
    └── module.py
```

Modules Python

- La Bibliothèque standard

- La bibliothèque standard contient les modules installés avec python

- Ses modules variés donnent au langage son caractère « multi-usage »

  https://docs.python.org/fr/3/library/index.html

- Exemples :

  - Expressions régulières : **re**
  - Fonctions mathématiques : **math**
  - Gestion des dates et heures : **datetime**
  - Internationalisation : **locale, gettext**
  - Intéraction avec l'OS : **sys, os, io**
  - Formats de fichiers : **csv, xml, json**
  - Formats de compression : **zlib**
  - Interfaces graphiques : **tkinter**
  - ...

DAWAN - Reproduction interdite

Modules Python

- Exemple : Datetime

- Permet de manipuler les dates

```
from datetime import *

datetime(annee, mois, jour, heure, minute, seconde, microseconde, fuseau horaire)
```

- Seuls l’année, le mois et le jour sont obligatoires

Modules Python

- Datetime : utilisations

IPython: Documents/formation_python

```
In [1]: from datetime import *

In [2]: datetime(2019, 8, 5)
Out[2]: datetime.datetime(2019, 8, 5, 0, 0)

In [3]: maintenant = datetime.now()

In [4]: maintenant.year, maintenant.month, maintenant.day
Out[4]: (2019, 8, 1)

In [5]: maintenant.hour, maintenant.minute, maintenant.second
Out[5]: (16, 21, 0)

In [6]: maintenant.date(), maintenant.time()
Out[6]: (datetime.date(2019, 8, 1), datetime.time(16, 21, 0, 548731))
```

Modules Python

- Datetime : durées (Timedelta)

IPython: Documents/formation_python

```
In [2]: duree = datetime(2019, 7, 16) - datetime(2015, 7, 17)

In [3]: duree
Out[3]: datetime.timedelta(days=1460)

In [4]: duree.days
Out[4]: 1460

In [5]: duree2 = datetime(2019, 8, 5) - datetime.now()

In [6]: duree2
Out[6]: datetime.timedelta(days=3, seconds=27009, microseconds=582063)

In [7]: duree2.seconds
Out[7]: 27009

In [8]: cuisson_oeuf = timedelta(seconds=180)

In [9]: maintenant = datetime(2019, 8, 5, 16, 30)

In [10]: maintenant + cuisson_oeuf
Out[10]: datetime.datetime(2019, 8, 5, 16, 33)

In [11]:
```

DAWAN - Reproduction interdite

Modules Python

- Datetime : Formatages de Dates

IPython: Documents/formation_python

```
In [1]: import datetime

In [2]: datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S.%f')
Out[2]: '2019-08-01 16:58:25.335720'

In [3]: datetime.datetime.now().strftime('%Y/%m/%d')
Out[3]: '2019/08/01'

In [4]: date_object = datetime.datetime.strptime('Aug 5 2019 4:33PM', '%b %d %Y %I:%M%p')

In [5]: date_object
Out[5]: datetime.datetime(2019, 8, 5, 16, 33)
```

Modules Python

- Calendar

IPython: Documents/formation_python

```
In [1]: import calendar

In [2]: calendar.mdays
Out[2]: [0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]

In [3]: calendar.isleap(2016)
Out[3]: True

In [4]: calendar.MONDAY, calendar.TUESDAY, calendar.WEDNESDAY
Out[4]: (0, 1, 2)

In [5]: calendar.weekday(2019, 8, 5)
Out[5]: 0
```

Python Orienté Objet

- Paradigmes de programmation

- Modèles de pensée et d'écriture de code comme chemin du problème vers la solution
- Programmation **impérative** : **blocs d'instructions** => machine d'état
- Programmation **fonctionnelle** : **fonctions et valeurs** => S = f(P)
- Programmation orientée objet **« POO »** : **classes et d'objets** => assemblage de « légos »
- Les classes sont des définitions de **structures** comprenant :

  - des variables appelées **attributs**
  - des fonctions appelées **méthodes**

- Les objets sont des **instances** ou des réalisations de la classe, comme un ouvrage d'art l'est d'un plan => le **type** d'un objet est sa classe

Python Orienté Objet

- UML : Unified Modeling Language

- Langage de **modélisation graphique standardisé** : 14 types de diagrammes en 2.5

- Répartis sur 3 vues :
  - **Fonctionnelle :** acteurs et besoins
    - diagrammes de cas d'utilisation

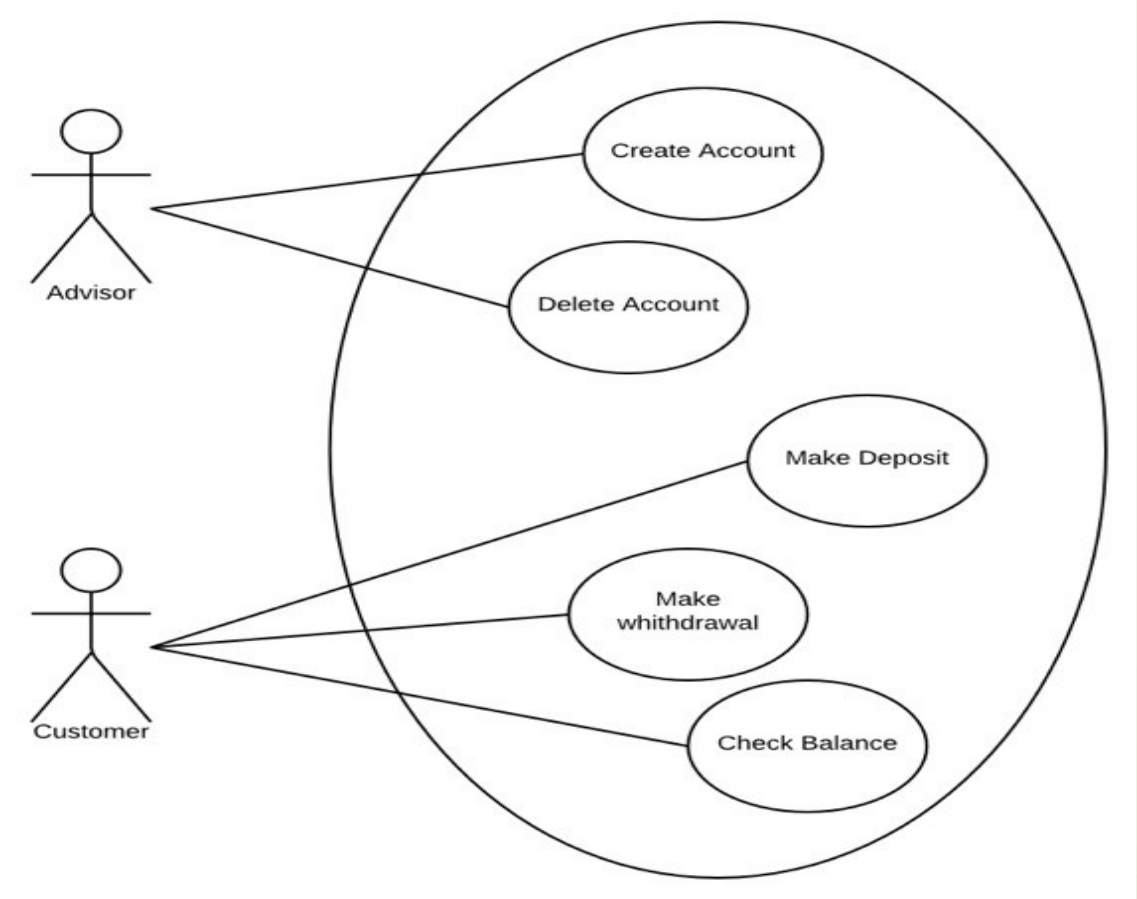
  - **Statique :** les structures de la solution et leur relations de dépendance
    - diagrammes de classe
    - diagrammes d'objet

  - **Dynamique :** comportement de ces structures dans le temps
    - diagrammes de séquence

DAWAN - Reproduction interdite

Python Orienté Objet

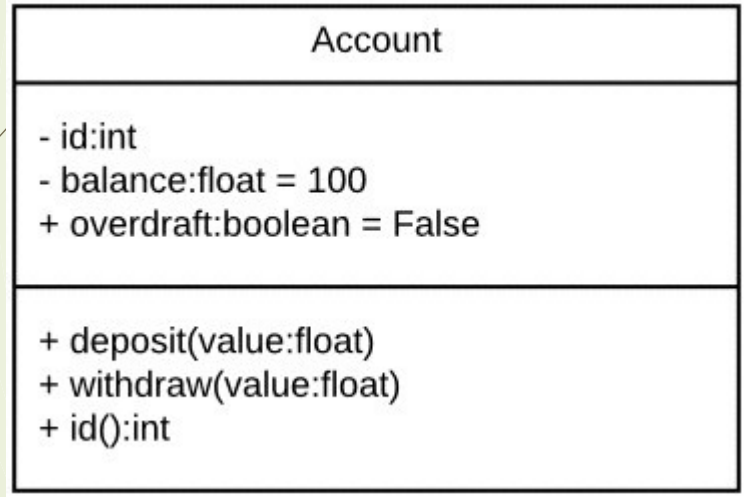
- UML : Diagrammes de cas d’utilisation



Python Orienté Objet

- Classes : Représentation et déclaration

  - Diagramme de classe UML / Création de classe Python



ATTRIBUTS

METHODES

```python
class Account:
    _id = 0
    _balance = 100.0
    overdraft = False
    def deposit(self, value):
        pass
    def widthdraw(self, value):
        pass
    def id(self):
        pass
```

Python Orienté Objet

- Objets : Représentation et instanciation

➤ Diagramme d'objet UML / Instanciation d'objet Python

Account : 555433921
- id = 555433921 - balance = 999.90 +overdraft = False

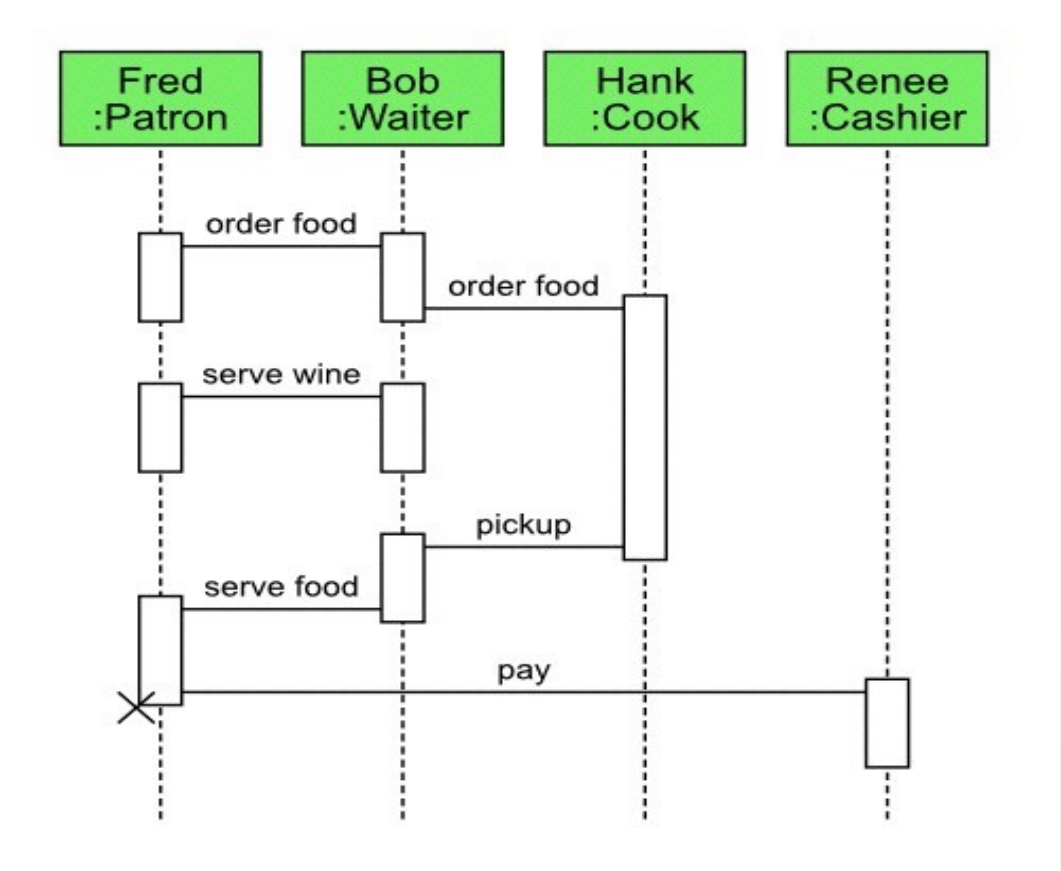
```
myAccount = Account()

myAccount._id = 555433921

myAccount._balance = 999.9
```

Python Orienté Objet

- UML : Diagrammes de séquence



Python Orienté Objet

- Visibilité des attributs et méthodes

- En python, l'accès aux attributs est **publique** : on peut les afficher et les affecter directement
- Par convention, un attribut précédé d'un underscore « _ » est **privé**
- On s'astreint à n'y accéder que par une **méthode publique => c'est l'encapsulation**
- Une **méthode privée** n'est appelable que depuis le corps d'une autre méthode
- **Self** est une référence d'instance : elle contient l'objet instancié
- Une méthode déclare toujours **self comme premier argument**

Python Orienté Objet

- Visibilité des attributs et méthodes

```
class Account:
    _id = 0
    _balance = 100.0
    overdraft = False
    def deposit(self, value):
        self._balance += value
    def widthdraw(self, value):
        self._balance -= value
    def id(self):
        return self._id
    def balance(self):
        return "{:.2f} €".format(self._balance)
```

```
myAccount = Account()

myAccount.deposit(899.9)

myAccount.balance()
'999.90 €'
```

Python Orienté Objet

- Docstrings, Introspection

  - **Les docstrings** sont disponibles pour documenter les classes, et nourrir la fonction **help()**
  - La fonction **isinstance(***obj***,** *ClassName***)** teste si l'objet ob est de classe ClassName
  - La fonction **dir(***obj***)** affiche les attributs et méthodes implicites de l'objet obj
  - Parmi ces méthodes on trouve :
    - **__init__** : initialise l'objet
    - __del__ : appelée avant la destruction de l'objet
    - __str__ : permet de convertir l'objet en chaine de caractère

Python Orienté Objet

- Constructeur

  - En redéfinissant **le constructeur __init__**, on peut initialiser l'objet dès l'instanciation

```
class Account:
    _id = 0
    _balance = 0
    overdraft = False
    def __init__(self, id, balance=100.0):
        self._id = id
        self._balance = balance
    def deposit(self, value):
        self._balance += value
    def widthdraw(self, value):
        self._balance -= value
    def id(self):
        return self._id
    def balance(self):
        return "{:.2f} €".format(self._balance)
```

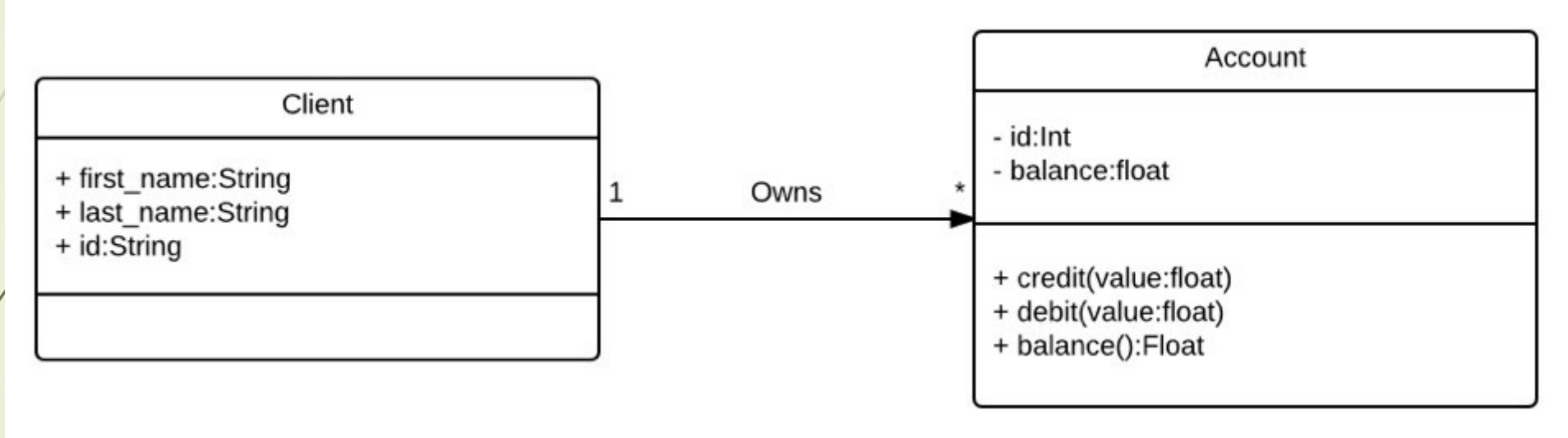
```
myAccount = Account(5559433, 999.9)

myAccount.id()
5559433

myAccount.balance()
'999.90 €'
```

Python Orienté Objet

- Associations entre classes : UML



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

Python Orienté Objet

- Associations entre classes : Python

```python
class Client:
    _id = 0
    _first_name = ""
    _last_name = ""
    def __init__(self, id, first_name, last_name):
        self._id = id
        self._first_name = first_name
        self._last_name = last_name
    def getLastName(self):
        return self._last_name
```

```python
ppClient = Client(1, "Paul", "Poule")

ppAccount = Account(134545, ppClient)

ppAccount.getClientName()
'Poule'
```

```python
class Account:
    _id = 0
    _balance = 0
    _owner = None
    overdraft = False
    def __init__(self, id, owner, balance=100.0):
        self._id = id
        self._balance = balance
        self._owner = owner
    def deposit(self, value):
        self._balance += value
    def widthdraw(self, value):
        self._balance -= value
    def id(self):
        return self._id
    def balance(self):
        return "{:.2f} €".format(self._balance)
    def getClientName(self):
        return self._owner.getLastName()
```

=> Couplage faible entre les classes : c'est l'injection de dépendance

Python Orienté Objet

- En Python, tout est objet !!!

```
x = -2

x.__abs__()
2

isinstance(x, int)
True

"bonjour".upper()
'BONJOUR'

isinstance("bonjour", str)
True
```

```
isinstance([1, 2], list)
True

class Test():
    pass


t = Test()

type(t)
__main__.Test
```

Python Orienté Objet

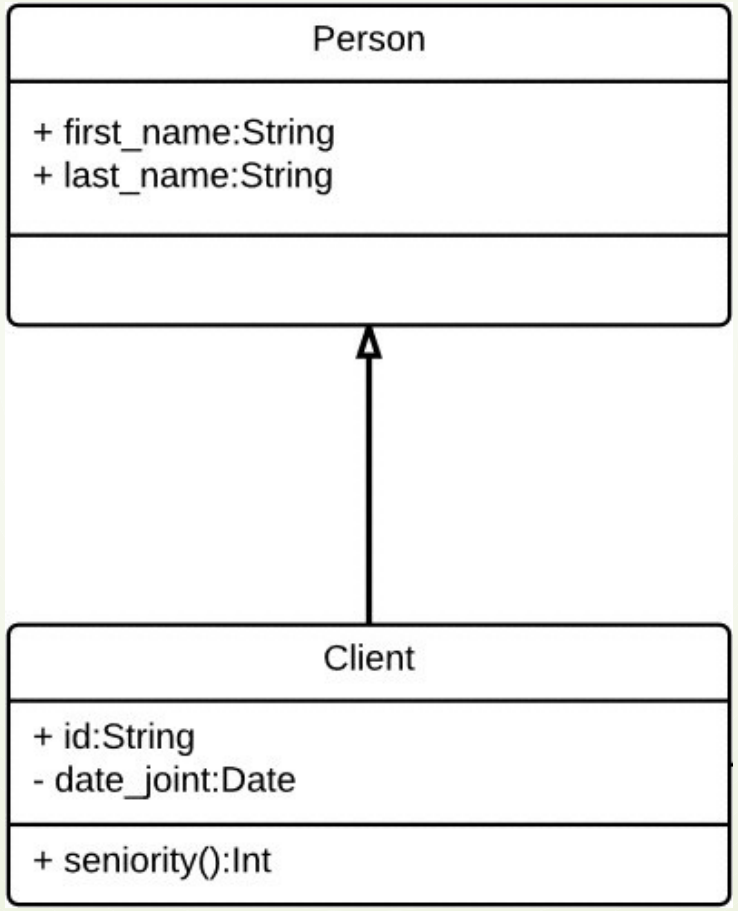
- L’héritage

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

DAWAN - Reproduction interdite

Python Orienté Objet

- L’héritage : UML



Python Orienté Objet

- L'héritage : Python

```
class Person:
    civility = ""
    first_name = ""
    last_name = ""
    def __init__(self, civility, first_name, last_name):
        self.civility = civility
        self.first_name = first_name
        self.last_name = last_name
    def getFullName(self):
        return self.civility.capitalize() + " " + \
               self.first_name.capitalize() + " " + \
               self.last_name.upper()
```

```
from datetime import *

boby = Client(1, "2019-08-07", "mr", "boby", "smith")

boby.date_joint.strftime("%Y-%m-%d")
'2019-08-07'

boby.getFullName()
'Mr Boby SMITH'
```

```
class Client(Person):
    id = 0
    date_joint = None
    def __init__(self, id, date_joint, civi, first, last):
        super().__init__(civi, first, last)
        self.id = id
        self.date_joint = datetime.strptime(date_joint, "%Y-%m-%d")
```

Python Orienté Objet

- L'héritage : Python

> Toutes les classes héritent de la classe **object**
> **L'héritage multiple** est disponible : Attention aux ambiguités !!!

```
class Person:
    id = 0
    def __init__(self, id):
        print("SetPersonid:", id)
        self.id = id
    def getId(self):
        print("GetPersonid:", self.id)
        return self.id
```

```
class Employee:
    id = 0
    def __init__(self, id):
        print("SetEmployeeid:", id)
        self.id = id
    def getId(self):
        print("GetEmployeeid", self.id)
        return self.id
```

```
class Sailer(Person, Employee):
    def __init__(self, idP, idE):
        print(idP, idE)
        Person.__init__(self, idP)
        Employee.__init__(self, idE)
    def getId(self, type_id):
        if type_id == "p":
            "Person"
            return Person.getId(self)
        "Employee"
        return Employee.getId(self)
```

```
In [4]: s = Sailer(1, 2)
1 2
SetPersonid: 1
SetEmployeeid: 2

In [5]: s.getId("p")
GetPersonid: 2
Out[5]: 2
```

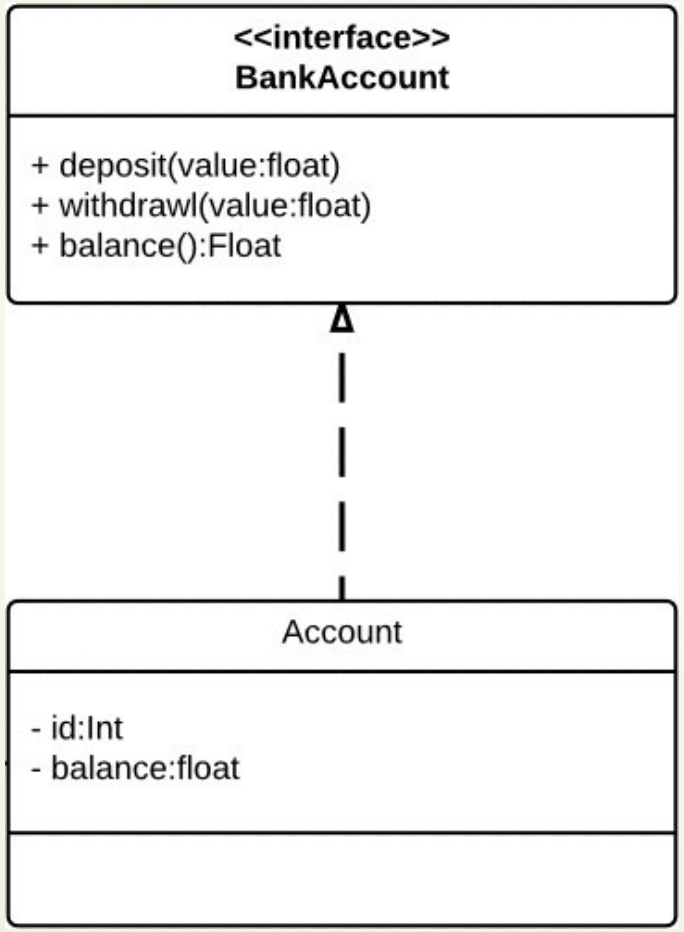
Python Orienté Objet

- Les « interfaces »

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

Python Orienté Objet

- Les « interfaces »



Python Orienté Objet

- Les itérateurs

- Sont des classes qui implémentent les méthodes implicites **__init__, __iter__ et __next__**

- Peuvent être utilisés avec iter() et next(), et dans les boucles for

```
class PowTwo:
  def __init__(self, max = 0):
    self.max = max
  def __iter__(self):
    self.n = 0
    return self
  def __next__(self):
    if self.n <= self.max:
      result = 2 ** self.n
      self.n += 1
      return result
    else:
      raise StopIteration
```

```
In [2]: a = PowTwo(4)

In [3]: i = iter(a)

In [4]: next(i)
Out[4]: 1

In [5]: next(i)
Out[5]: 2

In [6]: next(i)
Out[6]: 4
```

```
In [10]: for pt in PowTwo(5):
    ...:     print(pt)
    ...:
1
2
4
8
16
32
```

Python Orienté Objet

- Duck typing et polymorphisme

  - « Si çà vole comme un canard, cancane comme un canard, et nage comme un canard, alors cela doit être un canard » : Penser une fonction ayant un comportement cohérent avec plusieurs types de données

IPython: home/matthieu

```
In [1]: def fois3(x):
   ...:     return x * 3
   ...:

In [2]: fois3(9)
Out[2]: 27

In [3]: fois3("quoi ?")
Out[3]: 'quoi ?quoi ?quoi ?'
```

Python Orienté Objet

- Duck typing et polymorphisme
  - Le polymorphisme est le duck typing appliqué aux objets

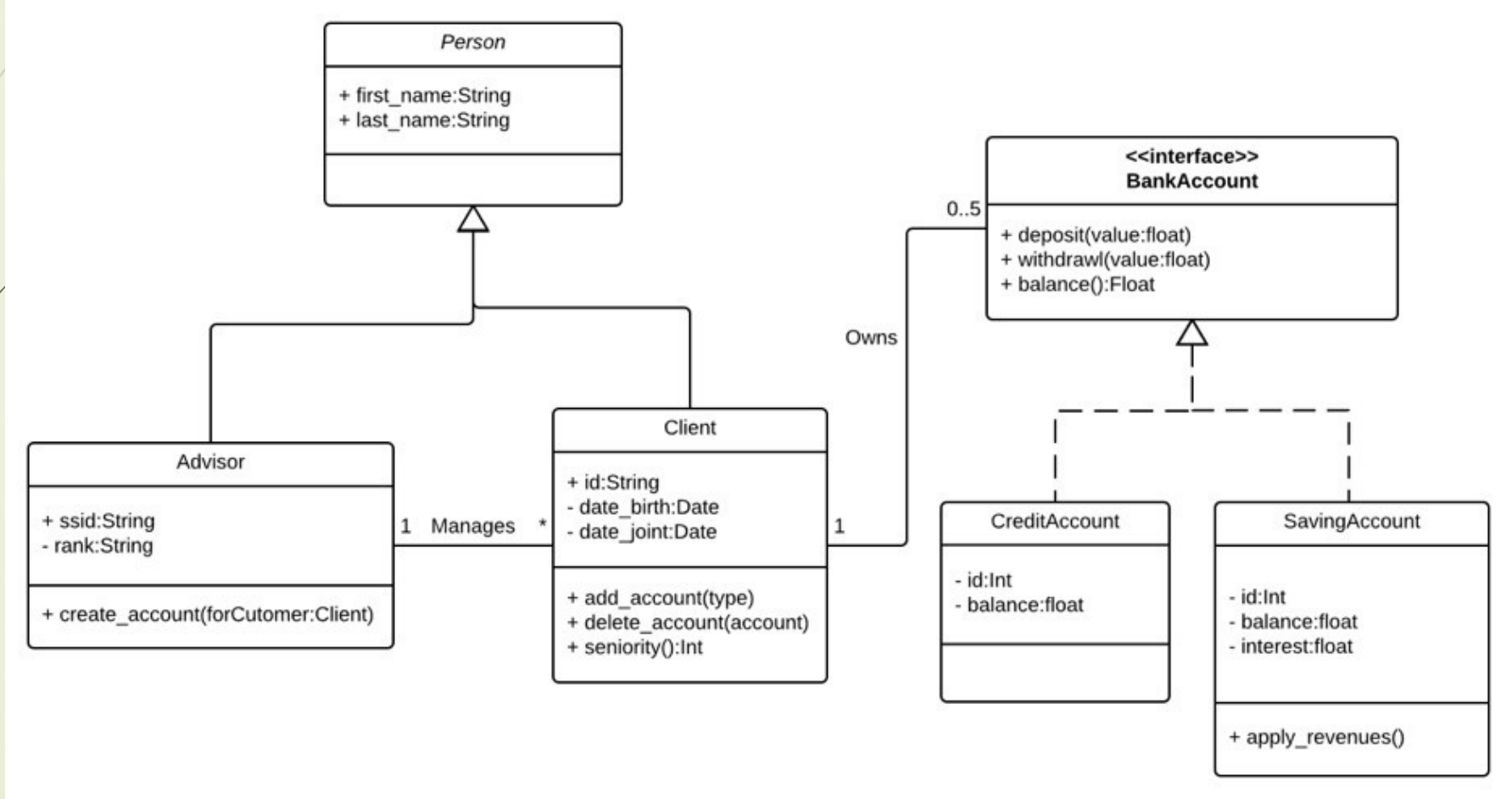
```
class Temp:
    def __init__(self, value, unit='C'):
        self.value = float(value)
        self.unit = unit
    def __float__(self):
        if self.unit == 'C':
            return self.value
        if self.unit == 'K':
            return self.value - 273.15
        if self.unit == 'F':
            return (self.value - 32) * 5/9
    def __repr__(self):
        return "{} {}".format(self.value, (self.unit != 'K')*'°'+self.unit)
```

```
t1, t2, t3 = Temp(5), Temp(3, 'K'), Temp(83, 'F')

t1, t2, float(t2), t3, float(t3)
(5.0 °C, 3.0 K, -270.15, 83.0 °F, 28.3333333333333332)
```

Python Orienté Objet

- UML : exemple complet



Les exceptions python

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

DAWAN - Reproduction interdite

Les exceptions python

- Exception non gérées

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

ModuleNotFoundError: No module named 'nimportequoi'

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

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

KeyError: 'nope'
```

Les exceptions python

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

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

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

Les exceptions python

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

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

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

Les exceptions python

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

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

Les exceptions python

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

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

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

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

Les exceptions python

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

IPython: Documents/formation_python

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

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

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

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

Les fichiers en Python

- Ouvrir un fichier
  - Les fichiers forment un type en python
  - On les manipule avec la fonction de base **open**

open(*path_to_filename*, [r || w || a] b ? **)**

  - r « read » : Ouvre le fichier en lecture seule
  - w « write » : Ouvre le fichier en écriture, écrase le contenu si déjà créé, crée le fichier sinon
  - a « append » : Ouvre le fichier en écriture, écrit à la fin du fichier
  - b « binary » : Placé après les précédents, ouvre le fichier en mode binaire (01)

Les fichiers en Python

- Manipuler un fichier : read, write, close

```
fichier_test = open("fichier_test.txt", "w")

fichier_test.write("Et l'unique cordeau des trompettes marines")
42

fichier_test.close()

fichier_test = open("fichier_test.txt", "r")

fichier_test.read()
"Et l'unique cordeau des trompettes marines"

fichier_test.close()
```

- **Toujours penser à refermer les fichiers quand on ne travaille plus avec**

Les fichiers en Python

- Manipuler un fichier : readlines, writelines

```
fichier_test = open("fichier_test.txt", "w")

strophe1 = """pigeon, oiseau à la grise robe,
dans l'enfer des villes,
"""

fichier_test.write(strophe1)
57

fichier_test.close()
```

```
In [9]: fichier_test = open("fichier_test.txt", "r")

In [10]: fichier_test.readlines()
Out[10]:
['pigeon, oiseau à la grise robe,\n',
 "dans l'enfer des villes,\n",
 'à mon regard, tu te dérobes,\n',
 'tu es vraiment le plus agile.\n']
```

```
fichier_test = open("fichier_test.txt", "a")

strophe2 = """à mon regard, tu te dérobes,
tu es vraiment le plus agile.
"""

fichier_test.write(strophe2)
59

fichier_test.close()
```

```
In [11]: fichier_test = open("fichier_test.txt", "a")

In [12]: fichier_test.writelines(["Mr Manatane", " Namur ", "1994\n"]
    ...: )

In [13]: fichier_test.close()

In [14]: fichier_test = open("fichier_test.txt", "r")

In [15]: fichier_test.readlines()
Out[15]:
['pigeon, oiseau à la grise robe,\n',
 "dans l'enfer des villes,\n",
 'à mon regard, tu te dérobes,\n',
 'tu es vraiment le plus agile.\n',
 'Mr Manatane Namur 1994\n']
```

DAWAN - Reproduction interdite

Les fichiers en Python

- Manipuler le curseur d'un fichier

- **Pointeur sur la position de la dernière écriture ou du mode d'ouverture**
- **tell()** donne la position du curseur dans le fichier
- **seek(** *offset, [whence]***)** place le curseur à la position *offset* à partir de *whence avec* :

  os.SEEK_SET = 0 => début du fichier
  os.SEEK_CUR = 1 => position courante du curseur => en binaire pour python 3.2+
- os.SEEK_END = 2 => fin du fichier => en binaire pour python 3.2+

```
fichier_test = open("fichier_test.txt", "r")

import os

fichier_test.read(7)
'pigeon,'

fichier_test.tell()
7
```

```
fichier_test = open("fichier_test.txt", "r")

fichier_test.seek(7, os.SEEK_SET)
7

fichier_test.tell()
7
```

Les fichiers en Python

- With : le « context manager »

- Ouvrir un fichier avec le mot-clé **with**
- Permet d'écrre un bloc qui **gère la fermeture du fichier** à la fin de son exécution

```
In [1]: with open("fichier_test.txt", "r") as f:
   ...:     text = f.read()
   ...:
```

Les fichiers en Python

- Fichier comme itérateur

- Un objet de type file possède un itérateur
- **next() :** retourne la ligne suivante

```
for line in f :
```

Les fichiers en Python

- Manipuler les chemins : module os

- On utilise le module **os**
- **os.remove(***path_to_file***)** : supprime un fichier
- **os.mkdir(***path_to_dir, mode=0o777***)** : crée un répertoire dans un chemin existant
- **os.makedirs(***path_to_dir***)** : crée un répertoire et les répertoires intermédiaires
- **os.removedirs(***path_to_dir***)** : supprime récursivement les répertoires si vides
- **os.rename(***path_to_old, path_to_new***)** : renomme ou déplace le fichier ou le répertoire
- **os.renames(***path_to_old, path_to_new***)** : idem avec création des répertoires intermédiaires

Les fichiers en Python

- Manipuler les chemins : module os

- **os.chdir(*path_to_dir*)** : change le répertoire de travail « cd »
- **os.getcwd()** : renvoie le chemin du répertoire de travail « pwd »
- **os.path.exists(*path*)** : True si le chemin existe, False sinon
- **os.path.isdir(*path*)** : True si le chemin est celui d' un répertoire, False sinon
- **os.path.isfile(*path*)** : Idem pour un fichier
- **os.listdir(*path_to_dir*)** : retourne la liste du contenu du répertoire
- **os.chmod(*path, mode*)** : change les permissions d'un fichier ou d'un répertoire (en octale)

Les fichiers en Python

- Manipuler les chemins : module shutil

- **shutil.move(***src, dest***)** : déplace ou renomme un fichier ou un répertoire « mv »
- **shutil.copy(***src, dest***)** : copie un fichier ou un répertoire
- **shutil.copy2(***src, dest***)** : Idem avec les métadonnées

Les fichiers en Python

- Manipuler les noms de fichiers

- **os.path.dirname(*path*)** : Retourne le chemin du répertoire contenant le chemin en paramètre
- **os.path.basename(*path*)** : Retourne le nom du fichier ou du répertoire sans le chemin
- **os.path.split(*path*)** : Retourne un tuple des deux précédents
- **os.path.splitext(*path*)** : Retourne un tuple pour obtenir l'extension

Les fichiers en Python

- Le format CSV

➢ **Comma Separated Values** : fichiers structurés en lignes / colonnes dits « à plat »

```
name;position;symbol;weight
Hydrogen;1;H;1.0079
Helium;2;He;4.0026
Lithium;3;Li;6.941
Beryllium;4;Be;9.0122
Boron;5;B;10.811
Carbon;6;C;12.0107
Nitrogen;7;N;14.0067
Oxygen;8;O;15.9994
Fluorine;9;F;18.9984
Neon;10;Ne;20.1797
Sodium;11;Na;22.9897
Magnesium;12;Mg;24.305
Aluminum;13;Al;26.9815
Silicon;14;Si;28.0855
Phosphorus;15;P;30.9738
Sulfur;16;S;32.065
Chlorine;17;Cl;35.453
Argon;18;Ar;39.948
Potassium;19;K;39.0983
Calcium;20;Ca;40.078
```

	A	B	C	D
1	name	position	symbol	weight
2	Hydrogen	1	H	1.0079
3	Helium	2	He	4.0026
4	Lithium	3	Li	6.941
5	Beryllium	4	Be	9.0122
6	Boron	5	B	10.811
7	Carbon	6	C	12.0107
8	Nitrogen	7	N	14.0067
9	Oxygen	8	O	15.9994
10	Fluorine	9	F	18.9984
11	Neon	10	Ne	20.1797
12	Sodium	11	Na	22.9897
13	Magnesium	12	Mg	24.305
14	Aluminum	13	Al	26.9815
15	Silicon	14	Si	28.0855
16	Phosphorus	15	P	30.9738
17	Sulfur	16	S	32.065
18	Chlorine	17	Cl	35.453
19	Argon	18	Ar	39.948
20	Potassium	19	K	39.0983
21	Calcium	20	Ca	40.078

DAWAN - Reproduction interdite

Les fichiers en Python

- Le format CSV

➢ On utilise le module csv et les classes **csv.reader et csv.writer**

```
In [1]: import csv

In [2]: with open("elements.csv", "r") as csvfile:
   ...:     csvReader = csv.reader(csvfile, delimiter=';', quotechar='"')
   ...:     for row in csvReader:
   ...:         print(", ".join(row))
   ...:
name, position, symbol, weight
Hydrogen, 1, H, 1.0079
Helium, 2, He, 4.0026
```

```
new_row = ["Scandium", 21, "Sc", 44.956]

with open("elements.csv", "a") as csvfile:
    csvWriter = csv.writer(csvfile, delimiter=';', quotechar='"', quotin
g=csv.QUOTE_MINIMAL)
    csvWriter.writerow(new_row)
```

Les fichiers en Python

- Le format JSON

- JavaScript Object Notation : syntaxe utile au stockage et à l'échange de données

```
{"elements":[
  { "position": 1, "name": "Hydrogen", "weight": 1.0079, "symbol": "H" },
  { "position": 2, "name": "Helium", "weight": 4.0026, "symbol": "He" },
  { "position": 3, "name": "Lithium", "weight": 6.941, "symbol": "Li" },
  { "position": 4, "name": "Beryllium", "weight": 9.0122, "symbol": "Be" },
  { "position": 5, "name": "Boron", "weight": 10.811, "symbol": "B" },
  { "position": 6, "name": "Carbon", "weight": 12.0107, "symbol": "C" },
  { "position": 7, "name": "Nitrogen", "weight": 14.0067, "symbol": "N" },
  { "position": 8, "name": "Oxygen", "weight": 15.9994, "symbol": "O" },
  { "position": 9, "name": "Fluorine", "weight": 18.9984, "symbol": "F" },
  { "position": 10, "name": "Neon", "weight": 20.1797, "symbol": "Ne" },
  { "position": 11, "name": "Sodium", "weight": 22.9897, "symbol": "Na" },
  { "position": 12, "name": "Magnesium", "weight": 24.305, "symbol": "Mg" },
  { "position": 13, "name": "Aluminum", "weight": 26.9815, "symbol": "Al" },
  { "position": 14, "name": "Silicon", "weight": 28.0855, "symbol": "Si" },
  { "position": 15, "name": "Phosphorus", "weight": 30.9738, "symbol": "P" },
  { "position": 16, "name": "Sulfur", "weight": 32.065, "symbol": "S" },
  { "position": 17, "name": "Chlorine", "weight": 35.453, "symbol": "Cl" },
  { "position": 18, "name": "Argon", "weight": 39.948, "symbol": "Ar" },
  { "position": 19, "name": "Potassium", "weight": 39.0983, "symbol": "K" },
  { "position": 20, "name": "Calcium", "weight": 40.078, "symbol": "Ca" }
]}
```

DAWAN - Reproduction interdite

Les fichiers en Python

- Le format JSON
  - On ultilise le module **json, les fonctions loads et dumps** pour travailler avec le format

IPython: Documents/formation_python

```
In [1]: import json

In [2]: with open("elements.json", "r") as j_file:
   ...:     j_string = j_file.read()
   ...:

In [3]: j_obj = json.loads(j_string)

In [4]: type(j_obj)
Out[4]: dict

In [5]: j_obj["elements"][3]["name"]
Out[5]: 'Beryllium'

In [6]: for el in j_obj["elements"]:
   ...:     print(json.dumps(el))
   ...:
{"position": 1, "name": "Hydrogen", "weight": 1.0079, "symbol": "H"}
{"position": 2, "name": "Helium", "weight": 4.0026, "symbol": "He"}
{"position": 3, "name": "Lithium", "weight": 6.941, "symbol": "Li"}
{"position": 4, "name": "Beryllium", "weight": 9.0122, "symbol": "Be"}
{"position": 5, "name": "Boron", "weight": 10.811, "symbol": "B"}
{"position": 6, "name": "Carbon", "weight": 12.0107, "symbol": "C"}
```

DAWAN - Reproduction interdite

Les fichiers en Python

- Le format XML

- **eXtended Markup Language** : langage de balise similaire à HTML

- Utile pour stocker et requêter des ensembles de données de moyenne taille sans SQL

```xml
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <elements>
    <element>
      <name>Hydrogen</name>
      <position>1</position>
      <symbol>H</symbol>
      <weight>1.0079</weight>
    </element>
    <element type="rare-gas">
      <name>Helium</name>
      <position>2</position>
      <symbol>He</symbol>
      <weight>4.0026</weight>
    </element>
    <element>
      <name>Lithium</name>
      <position>3</position>
      <symbol>Li</symbol>
```

Les fichiers en Python

- Le format XML
  - On ultilise le module **xml.etree.ElementTree** pour travailler avec le format

```
import xml.etree.ElementTree as ET

tree = ET.parse('elements.xml')

root = tree.getroot()

root.tag
'root'

root[0][1].attrib
{'type': 'rare-gas'}

root[0][1].attrib["type"]
'rare-gas'

root[0][2][0].text
'Lithium'
```

```
In [8]: for el in root:
   ...:     print(el.tag)
   ...:
elements

In [9]: for el in root.iter("element"):
   ...:     print(el.find("name").text)
   ...:
Hydrogen
Helium
Lithium
Beryllium
Boron
```

Les fichiers en Python

- Le format XML
  - La mise à jour se fait par manipulation de l'arbre et écriture

```
El  = ET.Element("element")

for k,v in {"name": "scandium", "position": "21", "weight": "44.956", "
ymbol": "Sc"}.items():
    Attr = ET.Element(k)
    Attr.text = v
    El.append(Attr)


ET.tostring(El)
b'<element><name>scandium</name><position>21</position><weight>44.956</
symbol>Sc</symbol></element>'

root[0].append(El)

tree.write("elemnts.xml")
```

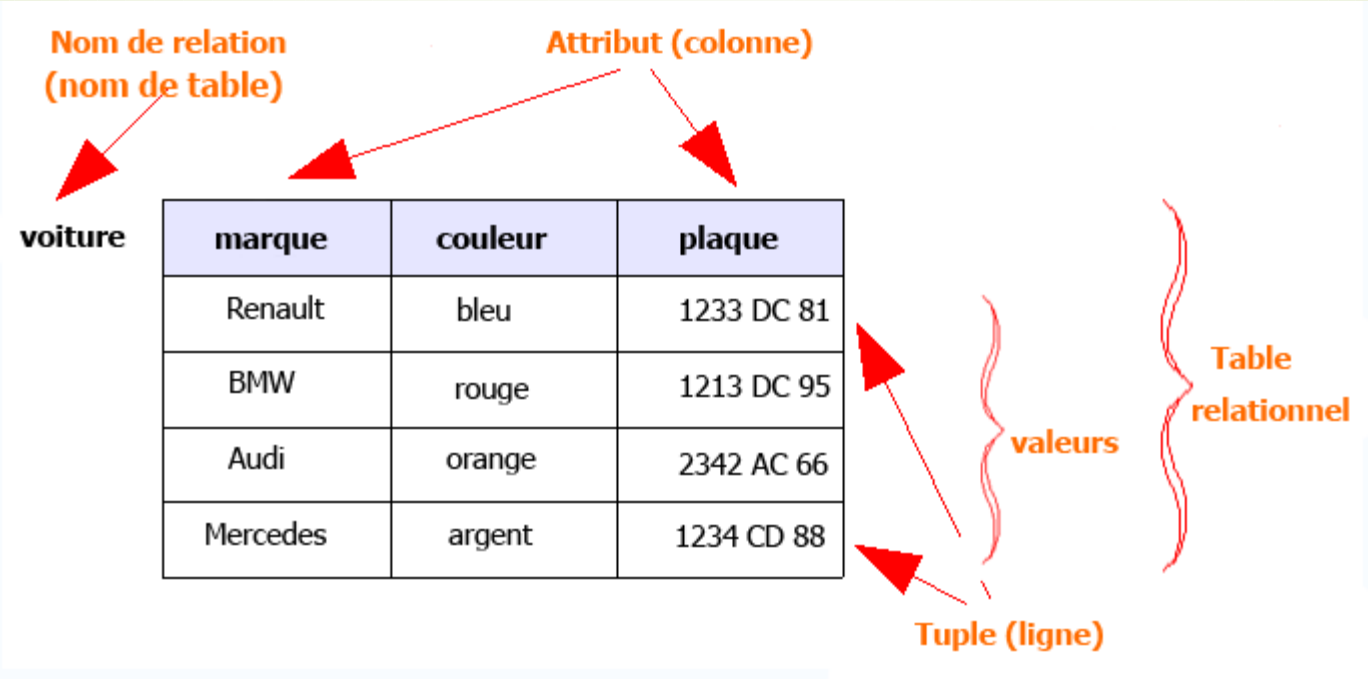
Python : bases de données

- Bases de données relationnelles

- Une base de donnée relationnelle est un **ensemble de fichiers**
- Ces fichiers organisent des données dans des tableaux à deux dimensions : **les tables**
- Les tables contiennent des **enregistrements (tuples!)** divisés en **champs nommés**
- Les champs forment des **colonnes** par nom de champs
- Les tables peuvent être apparentées par des **relations entre champs**
- Les bases de données relationnelles sont accessibles par des **requêtes SQL sur un serveur**
- Ex : **mysql/Mariadb, postgresql, oracle, IBM db2, sqlite3**
- En python sqlite3 est disponible dans la librairie standard, les autres par pip install

Python : bases de données

- Bases de données relationnelles



Python : bases de données

- SQL : création de tables

```
CREATE TABLE IF NOT EXISTS proprietaires (
    plaque CHAR(10) NOT NULL,
    nom VARCHAR(100) NOT NULL,
    prenom VARCHAR(100) NOT NULL,
    PRIMARY KEY (plaque)
)  ENGINE=INNODB;

CREATE TABLE IF NOT EXISTS voitures (
    voiture_id INT AUTO_INCREMENT,
    marque VARCHAR(100),
    couleur VARCHAR(100) NOT NULL,
    plaque CHAR(10) NOT NULL,
    create_date DATE CURRENT_TIMESTAMP,
    description TEXT,
    PRIMARY KEY (voiture_id)
    FOREIGN KEY proprietaires_plaque (plaque)
REFERENCES proprietaires(plaque)
)  ENGINE=INNODB;
```

Python : bases de données

- SQL : insertion de données

```sql
INSERT INTO proprietaires (plaque, nom, prenom) VALUES
  ('1233 DC 81', 'Martin', 'Rémy'),
  ('1213 FC 98', 'Benhamicha', 'Hakim'),
  ('2342 AC 43', 'Alloun', 'Jérémie')

INSERT INTO voitures (voiture_id, marque, couleur, plaque) VALUES
  (1, 'Renault', 'bleu', '1233 DC 81'),
  (2, 'BMW', 'rouge', '1233 DC 81'),
  (3, 'Audi', 'gris', '1233 DC 81'),
```

DAWAN - Reproduction interdite

Python : bases de données

- SQL : requêtes sur les données

```
SELECT voiture_id, marque FROM voitures WHERE couleur = 'bleu'

UPDATE voitures SET couleur = 'rouge' WHERE marque = 'Renault'

SELECT p.nom, p.prenom, v.marque, v.plaque
FROM proprietaires p JOIN voitures v ON p.plaque = v.plaque
ORDER BY v.plaque DESC
```

Python : bases de données

- DBAPI : Principe

- Interface d'accès à la plupart des serveurs de base de données selon les étapes suivantes :
- **Établir une connexion**
- **Créer un curseur** et lui attribuer une ou plusieurs requêtes
- **Exécuter** la ou les requêtes sur le curseur
- **Itérer** sur les éléments retournés : **FETCH**
- **Fermer la connexion**

Python : bases de données

- Sqlite3

- base de données **dans un unique fichier sur le disque**, sans accès serveur :
- **Pas d'utilisateur**
- **Disponible dans la lib standard**
- Lors de la connexion, si la base n'existe pas, elle est créée
- **On peut également travailler en mémoire : « :memory: »**

Python : bases de données

- Sqlite3 : connection

```
In [1]: import sqlite3 as lite

In [2]: conn = lite.connect('testdb.db')

In [3]: with conn:
   ...:     cursor = conn.cursor()
   ...:     cursor.execute("SELECT SQLITE_VERSION()")
   ...:     row = cursor.fetchone()
   ...:     print('server version', row[0])
   ...:
server version 3.27.2
```

Python : bases de données

- Pymysql
  - base de données de production open source la plus utilisée
  - **Disponible via pip install**
  - **l'objet de connection retourne directement un curseur**

```
#!/usr/bin/python3

import pymysql

db = pymysql.connect("localhost","testuser","test123","TESTDB" )

db.executescript("""
  CREATE TABLE IF NOT EXISTS voitures (
    voiture_id INT AUTO_INCREMENT,
    marque VARCHAR(100),
    couleur VARCHAR(100) NOT NULL,
    plaque CHAR(10) NOT NULL,
    create_date DATE CURRENT_TIMESTAMP,
    description TEXT,
    PRIMARY KEY (voiture_id)
    FOREIGN KEY proprietaires_plaque (plaque)
  REFERENCES proprietaires(plaque)
)  ENGINE=INNODB;
""")
```

Python : bases de données

- dbapi : créer une table

```
In [1]: import sqlite3 as lite

In [2]: conn = lite.connect(":memory:")

In [3]: cur = conn.cursor()

In [4]: cur.execute("CREATE TABLE IF NOT EXISTS proprietaires ( \
   ...:     plaque CHAR(10) NOT NULL, \
   ...:     nom VARCHAR(100) NOT NULL, \
   ...:     prenom VARCHAR(100) NOT NULL, \
   ...:     PRIMARY KEY (plaque))")
Out[4]: <sqlite3.Cursor at 0x7f75ac7521f0>
```

Python : bases de données

- dbapi : requêtes paramétrées

```
In [5]: conn = lite.connect('testdb.db')

In [6]: plaque, nom, prenom = "1233 DC 81", "Martin", "Rémy"

In [7]: with conn:
   ...:     cur = conn.cursor()
   ...:     cur.execute("INSERT INTO proprietaires (plaque, nom, prenom) \
   ...:     VALUES (?, ?, ?)", (plaque, nom, prenom))
   ...:     cur.execute("INSERT INTO proprietaires (plaque, nom, prenom) \
   ...:     VALUES (:plaque, :nom, :prenom)", {"plaque": "1213 FC 98", "nom": "B
   ...: enhamicha", "prenom": "Hakim"})
   ...:
```

Python : bases de données

- dbapi : « fetch » des enregistrements

- **fetchone :** récupération de l'enregistrement suivant
- **fetchmany([size]) :** récupération de size enregistrements suivants
- **fetchall() :** récupération de tous les enregistrements

```
In [9]: with conn:
   ...:     cur = conn.cursor()
   ...:     cur.execute("SELECT * FROM proprietaires")
   ...:     rows = cur.fetchall()
   ...:     print(rows)
   ...:
[('1233 DC 81', 'Martin', 'Rémy'), ('1213 FC 98', 'Benhamicha', 'Hakim')]
```

Python : bases de données

- dbapi : les transactions

- **commit :** valide les requêtes au bout d'un bloc **try : automatique dans un bloc with**
- **rollback : annulation** des requêtes dans des blocs except

```
In [1]: import sqlite3 as lite

In [2]: conn = lite.connect('testdb.db')

In [3]: try:
   ...:     with conn:
   ...:         conn.execute("INSERT INTO proprietaires (plaque, nom, prenom) \
   ...:           VALUES ('1233 DC 81', 'Martin', 'Rémy')")
   ...: except lite.IntegrityError:
   ...:     conn.rollback()
   ...:     print("couldn't add Martin twice!")
   ...:
couldn't add Martin twice!
```

Python : TKinter

- Présentation

- Bibliothèque graphique intégrée de base à Python
- Basée sur **Tk** (librairie c)
- L'avantage est la **portabilité**
- **https://docs.python.org/3.6/library/tkinter.html**

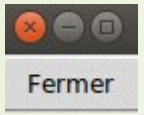
```
from tkinter import *

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

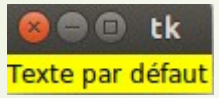
Python : TKinter

- widgets

Button

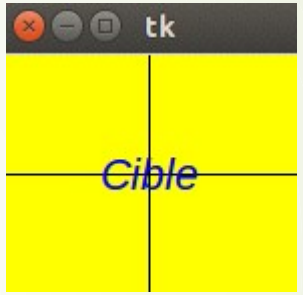


Label

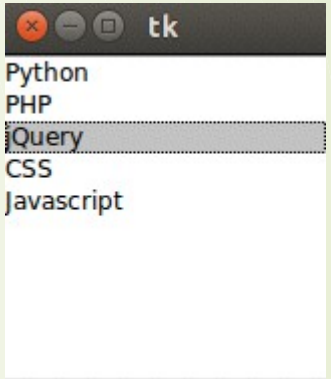


Entry

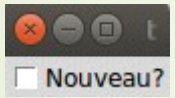
Canvas



Listbox

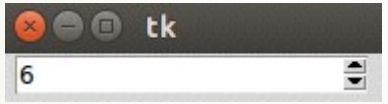


Checkbutton

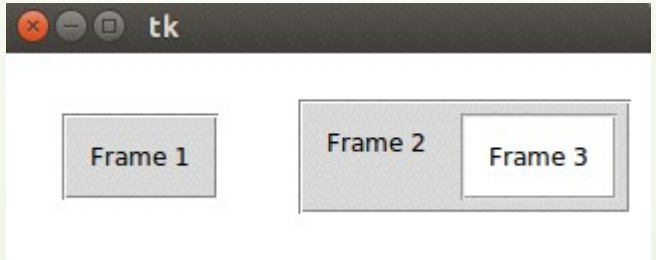


Scale

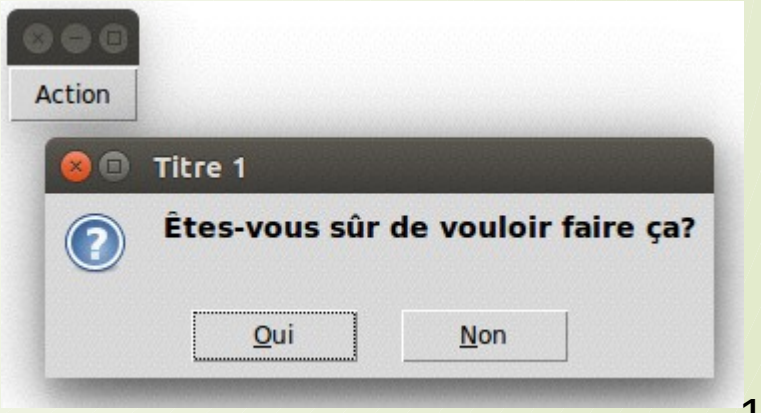
Spinbox



Frame



Messagebox



Python : TKinter

- Implémenter un widget

- Créer l'objet
- **Le premier paramètre est l'élément conteneur** (la fenetre ou autre conteneur)
- Les autres paramètres sont nommés (optionnels ou pas en fonction de l'élément)

text	texte du widget	pady	marge interne en y
foreground(fg)	couleur du texte	width	largeur en taille de police
activeforeground	idem quand selectionné	height	hauteur en taille de police
background(bg)	couleur de fond		
activebackground	idem quand selectionné		
padx	marge interne en x		

Python : TKinter

- Placer les composants : la méthode pack()

- Accepte un paramètre **side** : [« **TOP** » (défaut), « **BOTTOM** », « **LEFT** », « **RIGHT** »]
- Divise le conteneur en deux zones et place le widget du coté déclaré par side

```
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()
```

Python : TKinter

- Placer les composants : la méthode grid()

- Accepte les paramètres **column** et **row**
- **row=0, column=0** correspondent à **en haut à gauche**
- On choisit l'ordre d'affichage par l'ordre d'écriture des boucles
- **Incompatible avec pack() !!!**

```
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()
```

Python : TKinter

- Interactions : les boutons

➢ l'attribut **command** accepte des methodes internes à Tk ou des fonctions **« custom »**

```
from tkinter import *

fenetre = Tk()

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

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

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

fenetre.mainloop()
```

Python : TKinter

- Interactions : les entrées

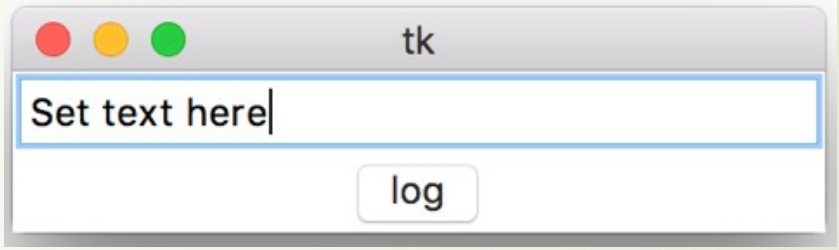
  - On utilise une classe Tk ad hoc pour intéragir avec la valeur saisie

```
from tkinter import *

fenetre = Tk()
entry_value = StringVar()
entry_value.set('Set text here')
Entry(fenetre, textvariable=entry_value, width=30).pack(side=TOP)

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

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



Python : TKinter

- Interactions : les checkboxes

➢ On utilise une classe Tk ad hoc pour intéragir avec la valeur « cochée »

```
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 {}".format(checkValue.get()))
  else:
    print('not Checked')

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

Python : TKinter

- Interactions : les listes de sélection

➢ On utilise une classe Tk ad hoc pour intéragir avec la liste des options sélectionnables

```
fenetre = Tk()

choices = Variable(fenetre, ('Java', 'Python', 'Swift'))
listbox = Listbox(fenetre, listvariable=choices, selectmode='single')
listbox.insert(END, 'Ruby')
listbox.insert(0, 'Pascal')
listbox.insert(1, 'Scala')
listbox.pack()

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

fenetre.mainloop()
```

Python : TKinter

- Macro composants

- 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

DAWAN - Reproduction interdite

Python : TKinter

- Macro composants : Exemple de classe

```
class UserInfo(Frame):
  def __init__(self, master=None, cnf={}, **kw):
    super().__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')
```

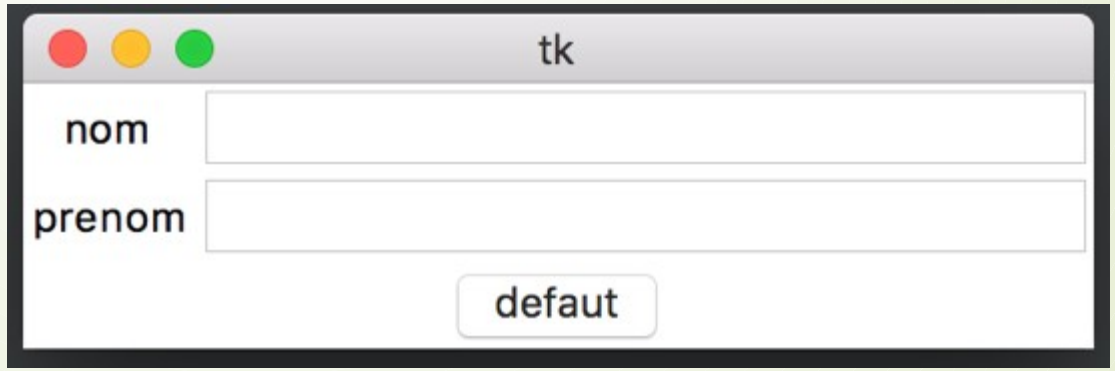
Python : TKinter

- Macro composants : Instanciation

```
from tkinter import *

fenetre = Tk()
fenetre.title('Demo')
user_frame = UserInfo(fenetre)
user_frame.pack()

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



Python : Les logs

- Enjeux

- Suivi de l'exécution d'un programme, sans l'interrompre
- Pour un programme en CLI, **print()** est à privilégier
- Le module standard **logging** permet un paramétrage fin de la journalisation
- Il met à disposition plusieurs niveaux d'alerte croissants, et un seuil d'écriture configurable :
  - **DEBUG :** information verbeuse à finalité de diagnostic
  - **INFO :** information de **franchissement d'étape** dans l'exécution
  - **WARNING :** quelquechose de **non bloquant** ne s'est pas déroulé comme prévu
  - **ERROR :** message d'**erreur bloquante**, dans un bloc except par exemple
  - **CRITICAL :** idem pour une erreur **bloquante et préjudiciable** à l'activité

Python : Les logs

- Module logging : les fonctions et la configuration

- 5 fonctions permettent d'émettre des messages reprenant les noms des niveaux d'alerte :
  - **debug**(*msg*), **info**(*msg*), **warning**(*msg*), **error**(*msg*) et **critical**(*msg*)
- **basicConfig**(*path*, *level=logging.WARNING*, […]) configure le fichier de logs et le seuil d'écriture

```
import logging
logging.basicConfig(filename='program.log',level=logging.INFO)
logging.debug("This message won't go to the log file")
logging.info('This one will')
logging.warning('So does that one')
```

Python : Les logs

- logging et modules importés

  - Si basicConfig s'exécute en début de programme, les modules importés peuvent utiliser logging

```
import logging
import mylib

def main():
    logging.basicConfig(filename='program.log', level=logging.INFO)
    logging.info('Started')
    mylib.do_something()
    logging.info('Finished')

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

```
import logging

def do_something():
    logging.info('Doing something')
```

Python : Les logs

- Formatage des messages

- Par l’injection de variables

```
logging.warning('%s of the %s', 'Beware', 'Dawg!')
```

- Par l’injection de paramètres depuis la configuration : **%(levelname)s, %(asctime)s, ...**

```
logging.basicConfig(format='%(asctime)s %(message)s', datefmt='%Y/%m/%d %I:%M’)

2019/08/08 08:48 Beware of the Dawg!
```

Python : requêtes HTTP

- Principe

- **HTTP: Hyper Text Transfer Protocol – protocole historique de communication sur le web**

- Basé sur des requêtes sur des **URLS**, avec échange d'entêtes et téléchargement d'une réponse

- Des gammes de code réponse dont les plus connus sont :

  - **200 :** la requête à été traitée
  - **301 :** idem mais l'url a été redirigée
  - **403 :** accès refusé
  - **404 :** page non trouvée
  - **500 :** erreur serveur

- 4 modes principaux d'acquisition, dit **« verbes HTTP »** assurant une fonction sémantique

  - **GET :** Lecture - utilisation de paramètres « querystring »?param1=[]&param2=[] dans l'URL
  - **POST :** Création - ajout de données (formulaires) dans la requête pour créer une ressource
  - **PUT :** Modification - idem mais pour mettre à jour une ressource
  - **DELETE :** suppression d'une ressource

DAWAN - Reproduction interdite

Python : requêtes HTTP

- Le module requests

  - Disponible via **pip install requests**

  - Requêtes basiques :

```
import requests

r = requests.get('https://api.github.com/events')
r = requests.post('https://httpbin.org/post', data = {'key':'value'})
r = requests.put('https://httpbin.org/put', data = {'key':'value'})
r = requests.delete('https://httpbin.org/delete')
```

  - Requête GET avec paramètres querystring :

```
payload = {'key1': 'value1', 'key2': 'value2'}
r = requests.get('https://httpbin.org/get', params=payload)
```

Python : requêtes HTTP

- Le module requests : la réponse

- Tester le code réponse

```
r.status_code == requests.codes.ok  #200
```

- Corps de réponse et paramètres :

```
r.text          # réponse en mode texte
r.content       # réponse en mode bytes ( pour les images par exemple)
r.json()        # réponse json
r.encoding      # encodage de la réponse
```

Python : requêtes HTTP

- Le module requests : les entêtes

- Entêtes de la requête

```
headers = {'user-agent': 'my-app/0.0.1'}
r = requests.get(url, headers=headers)
```

- Entêtes de la réponse :

```
r.headers
>>> { 'content-encoding': 'gzip', 'transfer-encoding': 'chunked',
    'connection': 'close', 'server': 'nginx/1.0.4', 'x-runtime': '148ms',
    'etag': '"e1ca502697e5c9317743dc078f67693f"',
    'content-type': 'application/json' }
```

Python : requêtes HTTP

- Le module requests : Upload d’un fichier

➢ Requête POST multipart :

```
files = {'file': open('example.xls', 'rb')}
r = requests.post(url, files=files)
```