

PYTHON

I. Introduction

historique

- Créé en 1989 par **Guido van Rossum**
- 1991: première version publique (0.9.0)
- 2000: python2: langage orienté script
- 2001: Fondation Python
- 2005: G. Van Rossum rejoint Google
- 2008: Python3: refonte fonctionnelle
- 2020: G. Van Rossum rejoint Microsoft



python sert à tout !

- **entreprises utilisant python**
- Dev Web: *Django, Odoo, Flask, FastAPI*
- Data Science: *Anaconda (Pandas, Numpy, Scipy, Sk-Learn, ...)*
- DevOps: *Ansible, Salt*

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

Langage interprété

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

Langage de Haut Niveau

- Gestion automatique de la mémoire
- Langage fortement typé *pas de conversions explicites*
- Typage dynamique *variable non déclarées, référence*

II. Setup

installation de python

- Linux: **python3** installé par défaut
 - ex: debian 12 => *python3.11*
 - autre version via un dépôt tiers **exemple**
 - idem via les sources et gcc installé **ici**
- Windows: **<https://python.org>**

“ *Windows: ajouter python & pip au PATH !!* ”

Installation de pip

- PIP: gestionnaire de paquets python
 - dépôt principal: **<https://pypi.org>**
- Windows: auto. avec python
- Linux:

```
sudo apt-get update  
sudo apt-get install -y python3-pip
```


commandes pip génériques

```
# install système  
pip install <package[(==|~=|<|>)<version>]>  
# install utilisateur  
pip install --user <package>  
# upgrader un paquet  
pip install <package> --upgrade  
# désinstaller  
pip uninstall <package>  
# upgrader pip lui même  
pip install -U pip
```

gestion simple des dépendances

```
# voir les dépendances installées  
pip freeze  
# générer un fichier de dépendances  
pip freeze > requirements.txt  
# installer les dépendances depuis un fichier  
pip install -r requirements.txt
```

Virtualenv : environnement virtuel

- Intérêt: **isolation** des *versions* des *binaires* et des *dépendances*
- le "Venv" est un **dossier** comprenant les binaires de python et pip ainsi qu'un conteneur de dépendances

```
## installation du module venv (linux)
sudo apt-get install -y python3-venv
## création du venv
python -m venv <venv_dir_path>
## activation du venv: redirection du binaire python et pip
# vers le dossier venv (Linux)
source <venv_dir_path>/bin/activate | deactivate
# idem (Windows)
./<venv_dir_path>/Scripts/Activate.ps1 | deactivate
```

install VSCode + Jupyter

1. installer VSCode
2. ajouter les extensions *python, jupyter*



3. créer un fichier python *.py*

Espaces : 4 UTF-8 CRLF { } Python 3.12.0 64-bit

4. sélectionner le **venv** en cliquant sur la version dans le footer
5. installer Jupyter dans le venv

```
pip install jupyter
```

utiliser les cellules Jupyter dans VSCode

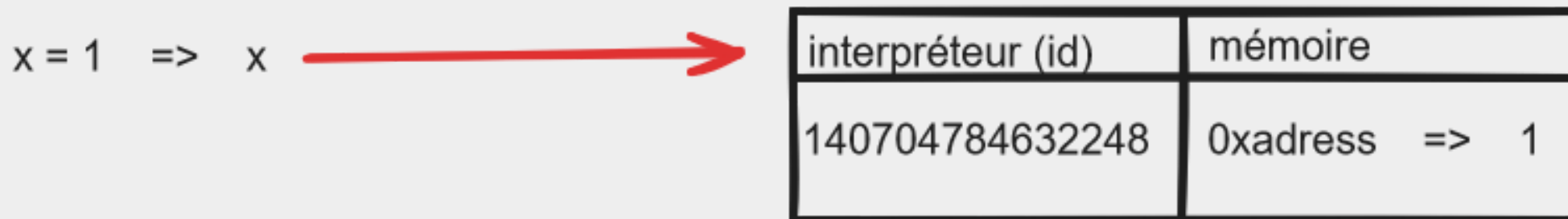
- une **cellule** est un ensemble de lignes de codes délimitées par `# %%`
 - les lignes d'une cellule seront exécutées dans la console *ipython* pilotée par *Jupyter*
 - `<Shift>+<Entrée>` pour exécuter
- “ *Les exécutions sont accumulées dans la console, peuvent créer des problèmes de gestions de variables*
Ne pas hésiter à redémarrer le kernel Jupyter pour en cas de pb ”

créer des notebooks Jupyter dans VSCode

- dans VSCode: `<Ctrl>+<Shift>+P + chercher jupyter`
- interface intégrée pour gérer auto. les cellules
- générer un fichier *.ipynb*

II. syntaxe python

variable



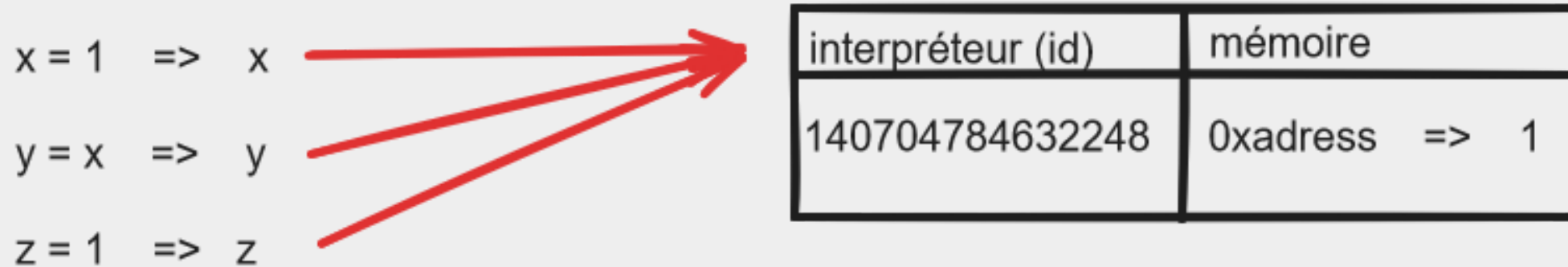
- une variable est une **référence** à une donnée dont la **valeur** est stockée en mémoire
- on crée une variable en *affectant* une *valeur* dans la variable de *nom* voulu `"="` est l'opérateur d'affectation

conventions de nommages : PEP8

ici

- Lettres seules, en minuscule: variables de type *compteur, indices*
 - **snake_case**: Lettres minuscules + « _ » : *tout sauf les classes*
 - **ALL_CAPS**: Lettres capitales + « _ »: *pseudo-constantes*
 - **CamelCase** : Noms de Classe (voir Python orienté objet)
 - Règle de nommage : "[a-zA-Z_][a-zA-Z0-9_]*"
- “ *Ne jamais créer une variable avec le nom d'un mot clé, module, fonction python !!*

passage par référence



- `y = x` : y est une **autre référence** à un emplacement mémoire existant
- `z = 1` : idem pour z car une autre variable a déjà référencé cette valeur **1** ici **x**

“ *ce dernier comportement vaut pour les types immutables cf infra* ” 17

fonctions de base: print

```
x = 10  
y = 20
```

```
## print: affiche la valeur d'une variable en tant que str  
print(x)
```

```
# affiche un nb quelconque de variable  
print(x, y)
```

```
# affiche une EXPRESSION python  
print(x + y + 30)
```

```
# changer les délimiteurs de valeur et de ligne  
# par défaut corresp. " " et "\n"  
print(x, y, sep="|", end=",")
```

notions d'instructions et expressions

- En programmation, une **instruction** désigne une *ligne de code syntaxiquement correcte*
 - Une expression désigne tout **élément de code évaluable**
 - les variables
 - les valeurs littérales `1, "hello", [1, 2, 3]`
 - toute association de variables et de littéraux, via des fonctions et des opérateurs ex: `x + abs(y) - 20`
- “ *Expression: tout ce qui est accepté dans la fonction print()* ”

fonctions de base: input

```
## input  
# 1/ bloque l'interpréteur  
# 2/ affiche une message introductif  
# 3/ permet d'écrire la valeur de retour au clavier  
  
_str = input("message intro")  
  
# 4/ la valeur de retour est une chaine de caractères STR
```

autres fonctions de base

Afficher la documentation sur une variable

`help(x)`

Retourner le type d'une variable

`type(x)`

Retourner les ATTRIBUTS INTERNES d'une variable

`dir(x)`

Retourner l'identifiant de l'emplacement mémoire référencé par la variable:

`id(x)`

Détruire une variable

`del x`

II. types "builtins"

```
# entier : int
obj = 10
# réel : float
obj = 3.14
# booléen : bool
obj = True # False
# chaîne de caractères : str
obj = "hello world"
# liste de valeurs hétérogènes ordonnées modifiables : list
obj = [10, 3.14, "hi"]
# idem, non modifiables : tuple
obj = (10, 3.14, "hi")
# dictionnaire: paires de clés / valeurs : dict
obj = {"nb": 10, 33: "trente trois"}
# valeur "nulle"
obj = None
```

conversions de type

- la grande majorité des conversions sont **explicites**, en utilisant les *fonctions de conversions*

```
# ERREUR de type
1 + "1"
# conversion d'entier
1 + int("1") # 2
# conversion de str => concaténation
str(1) + "1" # "11"
```

- il peut y avoir des conversions implicites

```
1 + 3.14 # 4.14
```

les types numériques

- `int, float, complex, bool`
- Fonctions de conversion `int(), float(), complex()`
- Représentations
 - binaire : `0b01011111, bin(95)`
 - octale : `0o755, oct(493)`
 - hexadécimale : `0x5f, hex(95)`
 - exponentielle : `3.14e-5`

opérateurs arithmétiques

Opérateur	Desc.
$x + y$	addition
$x - y$	soustraction
$x * y$	produit
x / y	division réelle
$x // y$	division entière (quotient)
$x \% y$	reste de la div. entière (modulo)
$-x$	opposé
$x ** y$	puissance (exposant)

opérateurs accumulatifs

Opérateur	Desc.
$x += y$	$x = x + y$
$x -= y$	$x = x - y$
$x *= y$	$x = x * y$
...	...

“ *pas d'opérateur $x++$ ou $++x$ en python !!!
 $x += 1$ à la place*

”

opérateurs logiques

Opérateur	Desc.
x or y	OU : x or y est vrai si x vrai ou y vrai
x and y	ET : x and y est vrai si x vrai et y vrai
not x	NON : not x est vrai si x est faux

valeurs booléennes des types builtins

- pour chaque type de données,
 - *une unique valeur* est considérée comme **False**
 - *toutes les autres valeurs* sont considérées comme **True**

valeurs booléennes

type	FAUX
int	0
float	0.
str	""
list	[]
tuple	()
dict	{}
NoneType	None

valeurs scalaires d'expressions booléennes

Expr.	val. bool.	val. scalaire	Desc.
33 or "ok"	True	33	1ère valeur vraie
None or []	False	[]	dernière valeur fausse
33 and "ok"	True	"ok"	dernière valeur vraie
None and []	False	None	1ère valeur fausse

```
value = <false_expression> or <default_value>
```

“ *on peut utiliser l'opérateur or pour établir une valeur par défaut en cas de valeur nulle !!!* ”

opérateurs de comparaison

Opérateur	Desc.
<code>x == y</code>	x égal à y <i>en valeur</i>
<code>x != y</code>	x différent de y
<code>x < y</code>	x strictement plus petit que y
<code>x <= y</code>	x plus petit ou égal à y
<code>x > y</code>	x strictement plus grand que y
<code>x >= y</code>	x plus grand ou égal à y
<code>x is y</code>	x référence y, <code>id(x) == id(y)</code>
<code>x is not y</code>	<code>id(x) != id(y)</code>

opérateur binaires

- opérateurs prennent leur sens en représentation binaires (0 et 1)

Opérateur	Desc.
$x \mid y$	OU
$x \wedge y$	OU exclusif
$x \& y$	ET
$x \ll n$	décalage à gauche, mul. par 2
$x \gg n$	décalage à droite, div. par 2
$\sim x$	NON

priorités des opérateurs

- Règles de priorités mathématiques
 - $** \rightarrow * \rightarrow + \dots$

Les parenthèses **()** ont la plus grande priorité

Op. *numériques* > *binaires* > *comparaison* > *logiques*

chaines de caractères : str

```
# single quotes
single_str = 'hello world'
# double quotes
double_str = "that's all folks !"
# désactiver le saut de ligne
long_command = "cmd sub_cmd \
    --option1 val1 \
    --option2 val2"

# incorporer le saut de ligne dans la valeur de str
paragraph = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
Ut enim ad minim veniam"""
```

listes : list

- liste de valeurs
 - hétérogènes: **de types !=**
 - ordonnées: **indexables**
 - modifiables: **mutables**

```
age = 33  
nom = "michel"  
lst = [nom, age, 175.6]
```

t-uples : tuple

- liste de valeurs
 - hétérogènes: **de types !=**
 - ordonnées: **indexables**
 - *non modifiables*: **immutables**

```
age = 33  
nom = "michel"  
tpl = (nom, age, 175.6)
```

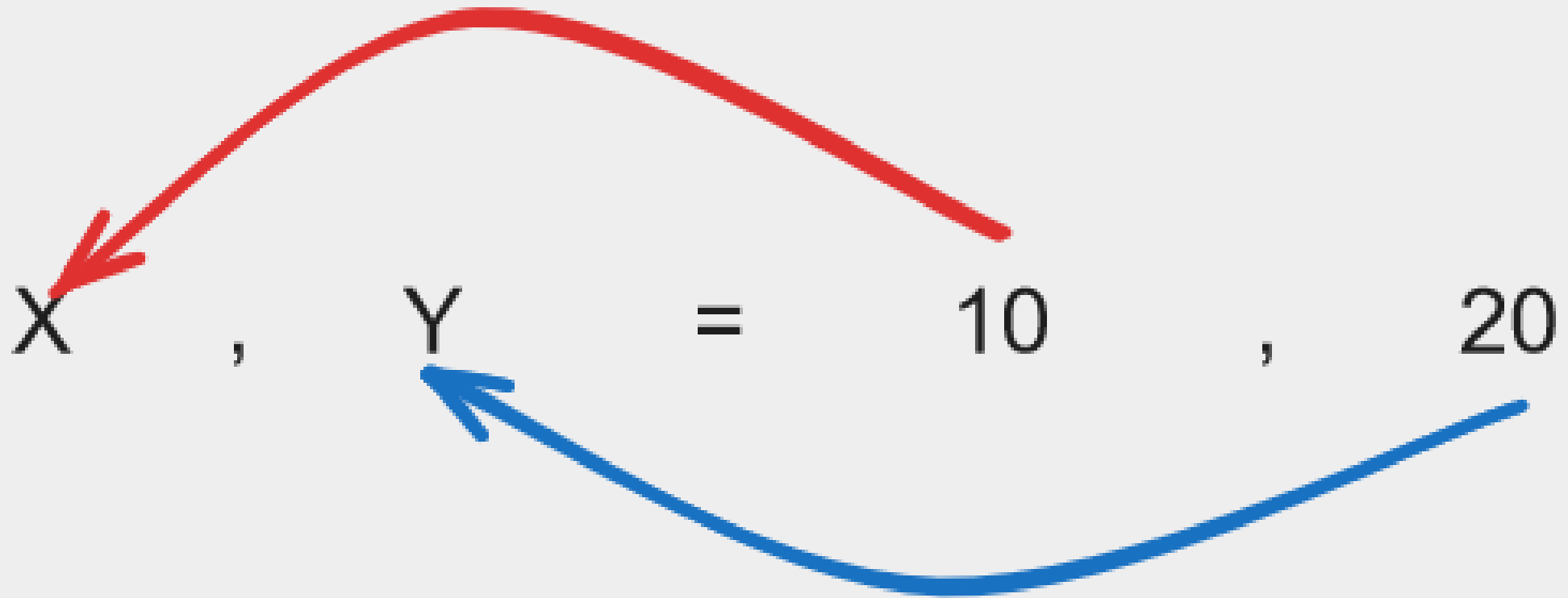
```
## CONVERSIONS  
tpl = list(tpl)  
lst = tuple(tpl)
```

"unpacking" : tuples implicites

```
lst = [33, "michel", 175.6]
# déchargement des n valeurs à droite dans n variables à gauche
age, nom, taille = lst
# même technique: affectations simultanées
x, y = 1, 2          # idem: x, y = (1, 2)

# ATTENTION
a, b = 0, 1
# différence entre
a = b
b = a + b
# et
a, b = b, a + b
```

unpacking : schema



types indexables

- utilisation de l'opérateur *"crochet"* `[i]` avec `i` entier
- retourne **la valeur à l'index `i`** de la variable
- types indexables: `str, list, tuple` dites **séquences**

```
word, lst = "anticonstitutionnellement", [1, 2, 3, 4, 5, "end"]
```

```
# index >= 0 : de gauche à droite : à partir de l'index 0
```

```
word[0], word[2], lst[0], lst[2]
```

```
# index < -1 : de droite à gauche : à partir de l'index -1
```

```
word[-1], word[-2], lst[-1], lst[-2]
```

```
tpl = (word, lst)
```

```
tpl[0][5], tpl[1][-1] # un opérateur peut s'effectuer sur une expression !
```

mutabilité vs immutabilité

- un type dit **mutable** permet de modifier la valeur d'une variable
 - *sans changer son identifiant mémoire*
 - autrement dit *sans affectation globale de la variable*
- un type donnée dit **immutable** ne permet pas ces modifs

```
lst, tpl, word = [-1, 2, 3], (-1, 2, 3), "bonjour"  
# list est MUTABLE  
lst[0] = 1 # OK, modification partielle, id(lst) inchangé  
# str et tuple sont IMMUTABLE  
word[0] = "B" # TypeError  
tpl[0] = -1 # TypeError  
word, tpl = "Bonjour", (1, 2, 3) # OK mais modifie id(word) et id(tpl)
```


"slicing" des séquences

- retourne la **sous séquence** délimitée par 2 indices

```
seq = "Hello World!"

# [index de départ COMPRIS] ou [le début] => index de fin NON COMPRIS
seq[0:5], seq[:5] # Hello

# index de départ COMPRIS => [index de fin NON COMPRIS] ou [la fin]
seq[6:12], seq[6:], seq[-6:] # World!

# idem avec un pas de prélèvement de 2 ici
seq[ 1:11:2 ] # el ol
seq[ ::-1 ] # INVERSE !
```

opérateurs sur les séquences

```
# concaténations
sentence = "un chat est" + " " + "un chat"
dizaine = [0,1,2,3,4] + [5,6,7,8,9] # [0,1,2,3,4,5,6,7,8,9]

# répétitions
locution = 3 * "rien " # rien rien rien
vecteur = 10 * [1] # [1,1,1,1,1,1,1,1,1,1]

# appartenance : évalue si
# un élément est dans la séquence
faux = 11 not in dizaine
# voire si une sous séquence est dans la séquence (spécifique str)
vrai = "chat" in sentence
```

fonctions globales sur les séquences

```
lst = [1, 2, 3, 4, 5, "six", "sept"]  
# longueur de la séquence  
len(lst) # 7  
# fonctions effectives sur des éléments de même type  
min(lst), max(lst), sum(lst) # Erreur: TypeError  
  
min(lst[:5]), max(lst[:5]), sum(lst[:5]) # 1, 5, 15
```

attributs internes des séquences

- `_str.index()`: l'opérateur `"."` permet d'appeler les attributs d'une variable

```
sentence = "un chat est un chat"  
# trouver l'index d'un élément ici du 1er "c"  
sentence.index("c") # 3  
# trouver l'index de la sous chaine (spécifique à str)  
sentence.index("chat") # 3  
# idem à partir de l'index 7 => 2ème occurrence de "chat"  
sentence.index("chat", 7) # 16  
## nb d'occurrence d'un élément  
sentence.count("chat") # 2
```

valeurs de retour des attributs internes

- les attributs internes des types **immuables** *int, float, str, tuples*
 - retournent une valeur transformée
 - mais ne peuvent pas modifier la variable
- pour les types **mutables** *list, dict*
 - certains modifient la valeur et retourne rien
 - certains modifient la valeur et retourne qqch
 - certains ne modifient pas la valeur et retourne la transformation

attributs internes spécifiques à str

- `dir(str)` : liste les *fonctions internes* utilisables sur le type str

```
# EXEMPLES
```

```
lucky_number, sentence = "777", "un|chat|est|un|chat"
```

```
# évalue si la str est composée de caractères uniquement numériques
```

```
lucky_number.isnumeric() # True
```

```
# découper les éléments d'un str selon un délimiteur
```

```
words = sentence.split("|") # ['un', 'chat', 'est', 'un', 'chat']
```

```
# soude les éléments d'une liste de str avec une str
```

```
" ".join(words) # "un chat est un chat"
```

présenter les résultats

```
prenom, nom, moyenne = "bob", "smith", 15.8875

# BAD PRACTICE : gestion des " " et les conversions
print("prénom: " + prenom + " , nom: " + nom + " , moyenne: " + str(moyenne))
# mieux mais BAD PRACTICE
print("prénom:", prenom, ", nom:", nom, ", moyenne:", moyenne)

## TEMPLATING
template = "prénom: %s, nom: %s, moyenne: %.2f" # old-school python2
print(template % (prenom, nom, moyenne))
template = "prénom: {fname}, nom: {name}, moyenne: {avg}" # python3 BEST PRACTICE
print(template.format(name=nom, fname=prenom, avg=moyenne))

## f-strings: injecter direct. des expressions python dans une str BEST PRACTICE
print(f"prénom: {prenom}, nom: {nom.upper()}, moyenne: {round(moyenne, 2)}")
```

attributs internes de pile de list

- `dir(list)` , ex: `lst = [2, 3, 5]`

attribut	effet	retour	lst
<code>lst.append(6)</code>	ajout à droite	None	[2,3,5,6]
<code>lst.insert(2,4)</code>	ajout à gauche de l'index 2	None	[2,3,4,5,6]
<code>lst.insert(0,1)</code>	ajout à gauche	None	[1,2,3,4,5,6]
<code>lst.pop()</code>	supprime à droite	6	[1,2,3,4,5]
<code>lst.pop(2)</code>	supprime à l'index	3	[1,2,4,5]
<code>lst.remove(5)</code>	supprime 1er occ. de la val.	None	[1,2,4]

transformation interne vs globale

- ex: `lst = [6, 5, 4]`

attribut	effet	retour	lst
<code>lst.extend([3,2,1])</code>	concatène	None	[6,5,4,3,2,1]
<code>lst + [3,2,1]</code>	cocnatène	[6,5,4,3,2,1]	[6,5,4]
<code>lst.sort()</code>	trie	None	[4,5,6]
<code>sorted(lst)</code>	trie	[4,5,6]	[6,5,4]
<code>lst.reverse()</code>	inverse	None	[4,5,6]
<code>lst[::-1]</code>	inverse	[4,5,6]	[6,5,4]

dictionnaires : dict

- paires de **clés / valeurs**
- les **clés** sont *uniques* et de types *immuables* "hashables"

```
## littéral
dico = {
    "key": "value",
    "other_key": 10,
    42: [1, 2, 3]
    3.14: "PI"
    (48.86, 2.33): "PARIS"
}
# conversion à partir d'une liste de tuples
dico = dict([("key", "value"), ("other": "...")])
dico = dict(key="value", other="...") # instantiation cf infra
```

dictionnaires : usages

- les valeurs sont référencées par les **clés != indexes**

```
dico["key"] # accès aux valeurs au moyen des clés
dico["unknown key"] # ERREUR KeyError
# MUTABLES
dico["key"] = "other value"
dico["new_key"] = "new value"
# opérateur IN avec les CLES
"key" in dico # True
# valeur d'une clé OU valeur par défaut SI clé absente
dico.get("key", "default") # "value"
dico.get("unknown key", "default") # "default"
# suppression de clés
dico.pop("key") # modifie et retourne la valeur
del dico["key"]
```

dictionnaires: transformations remarquables

```
# retourne la liste des clés  
list(dico)  
# retourne la liste des valeurs  
list(dico.values())  
# retourne la liste des tuples (clé, valeur)  
list(dico.items())  
  
# créer un dictionnaire à partir de deux listes : clés et valeurs  
keys, values = ["k1", "k2", "k3"], ["v1", "v2", "v3"]  
dict(zip(keys, values))
```

affectation de variables mutables

- l'affectation `=` effectue un passage par référence, i.e ajoute une autre référence sur un emplacement mémoire existant
- dans le cas de variables mutables *list ou dict* on peut avoir des effets de bords désagréables !

```
lst = [1, 2, 3]
lst2 = lst # passage par référence
lst.append(4) # modifie lst ET lst2 car
print(lst is lst2, lst2) # True, [1,2,3,4]
```

```
lst = [1,2,3]
lst3 = lst.copy() # ou lst[:] => "copie creuse" i.e indépendante en mémoire
lst.append(4) # modifie lst MAIS PAS lst3 car
print(lst is lst3, lst3) # False, [1,2,3]
```

III. structures de contrôles

principe

- Structure de code utilisant:
 - Des en-têtes terminés par ":"
 - Des **blocs de code** exécutés selon l'entête
- “ *En Python, les blocs sont définis par l'indentation (2 ou 4 espaces)* ”

IF : structure conditionnelle

```
# en-tête principale
if <expr.>:
    # bloc if: indentation, exécuté si <expr.> vraie
# fin du bloc if: revenir à l'indentation précédente
# en-têtes facultatifs
elif <alt_expr.>:
    # bloc elif #1: indentation, exécuté si <expr.> fausse
    # et <alt_expr.> vraie
# fin du bloc elif #1: revenir à l'indentation précédente
elif ...
# en-tête facultatif
else:
    # bloc else : indentation,
    # exécuté si les expressions précédentes sont fausses
# fin du bloc else: revenir à l'indentation précédente
```

opérateur ternaire

```
if cond:  
    valeur = <val_1>  
else:  
    valeur = <val_2>  
  
# opérateur ternaire: "sucre syntaxique"  
valeur = <val_1> if cond else <val_2>
```


syntaxe "switch"

- si le code a besoin de beaucoup structures **if**, **elif** et **else** pour exécuter des *instructions différentes* selon les valeurs d'*une variable*

```
match variable:
    case <val_1>:
        # bloc_1
    case <val_2>:
        # bloc_2
    case <val_3>:
        # bloc_3
    case _:
        # bloc par défaut
```

FOR: itération selon une séquence

- exécutions *successives* d'un bloc
- injectant *successivement* les valeurs d'une **séquence** de l'en-tête
- dans une **variable** de l'en-tête,
- à proprement parler, tout objet qu'on peut placer dans une boucle for sera désignée **itérable**

```
for elem in iterable:  
    # bloc for: utilisation de la variable elem  
# fin du bloc for: quand toutes exécutions successives du bloc for  
# ont été terminées
```

itérations sur des entiers: range

- par défaut la boucle **for** itère sur les *valeurs* d'un itérable
- pour créer un *compteur*, on aurait besoin a priori d'une liste d'entiers
- python utilise la fonction `range()`

```
# depuis 0 COMPRIS à la fin NON COMPRIS
for i in range(10): print(i) # [0,1,2,3,4,5,6,7,8,9]
# depuis début COMPRIS à la fin NON COMPRIS
for j in range(1, 11): print(j) # [1,2,3,4,5,6,7,8,9,10]
# depuis début COMPRIS à la fin NON COMPRIS en prenant un pas
for k in range(1, 11, 2): print(k) # [1,3,5,7,9]
# à rebours
for m in range(10, -1, -1): print(m) # [10,9,8,7,6,5,4,3,2,1,0]
```

itérations sur les indexes

- on peut utiliser `enumerate()` pour itérer à la fois sur *l'index et la valeur* d'un itérable
- les indexes commencent à partir de **0** par défaut

```
fruits = ["pomme", "banane", "abricot"]  
for i, fruit in enumerate(fruits):  
    fruits[i] = fruit.upper()  
print(fruits) # ["POMME", "BANANE", "ABRICOT"]
```

“ *iterate() permet de transformer l'itérable lui même pendant la boucle* ”

liste en intension

- on appelle l'**intension** en logique, la *définition* d'un concept
- une liste en intension est créé selon un algorithme définissant les valeurs

```
# je veux les expressions "f.upper()" pour "f" pris dans la liste "fruits"  
[ f.upper() for f in fruits ] # ["POMME", "BANANE", "ABRICOT"]
```

```
# idem avec filtre  
[ f.upper() for f in fruits if f[0] != "P" ] # ["BANANE", "ABRICOT"]
```

WHILE : itération selon une condition

```
## si '<expr.>' est vraie  
# on exécute le bloc while et on reteste '<expr.>'  
# TANT QUE <expr.> soit fausse !  
while <expr.>:  
    # bloc while
```

- sortie de la boucle
 1. soit le contenu du bloc rend `<expr.>` à False, => la boucle s'arrête à la fin du bloc
 2. soit `<expr.>` reste toujours vraie => **boucle infinie !!!**
 3. exécution d'une instruction `break` dans le bloc => la boucle s'arrête *immédiatement*

WHILE : exemple

```
# suite de Fibonacci < 100  
a, b = 0, 1  
while b < 100:  
    a, b = b, a + b
```

“ *REM: quelle est la valeur du "a" de droite dans un unpacking !* ”

manipuler une boucle

- dans une boucle `for` ou `while`
 - `break` : *interrompt le bloc* et sort de la boucle **immédiatement**
 - `continue` : *interrompt le bloc* et exécute **l'itération suivante**
- après une boucle `for` ou `while`
 - `else` : exécute un bloc si la boucle associée **s'est terminée sans `break`**
- l'instruction `pass` figure un bloc *qui ne fait rien* en dessous d'un en-tête

break, continue, else : exemple

```
from decimal import Decimal
prix = [19.89, 24.50, 2.328, -3.4]
for i, p in enumerate(prix):
    # prix négatif => STOP !!
    if p <= 0:
        print(f"prix {p} négatif !!")
        break
    d = Decimal(str(p))
    nb_decim = -1 * d.as_tuple().exponent
    # prix bien formé: RAS
    if nb_decim <= 2: continue
    # prix mal formé: on arrondit à 2 décimale
    prix[i] = round(p, 2)
# s'il n'y a pas de prix aberrant
else:
    print(f"tous les prix sont OK: {prix}")
```

itérations sur les dictionnaires

```
dico = { f"key_{i}": f"val_{i}" for i in range(1, 4) }  
# Iterations sur les clés  
for k in dico: print(k)  
# Iterations sur les values  
for v in dico.values(): print(v)  
# Iterations sur les paires / clés  
for k, v in dico.items(): print(k, v)
```

IV. Fonctions:

définition et appel

- fonction = bloc de code **nommé paramétrable et réutilisable**

```
## DEFINITION d'une fonction
# en-tête def <nom> (<params>): ensemble des paramètres = SIGNATURE
def ma_fonction():
    # bloc fonction
    print("coucou")
# fin du bloc fonction

## APPEL de la fonction ma_fonction
# => exécution du bloc fonction
ma_fonction()
```

valeurs de retour

- l'instruction `return <expr.>` va donner une valeur de retour à l'appel d'une fonction ou `None` si absente *procédure*

```
def ma_fonction():  
    print("coucou")  
    # return None  
def ma_return_fonction():  
    return "coucou"  
  
ret = ma_fonction()  
print(ret) # None  
# la valeur de retour "coucou" de la fonction  
# est affectée à la variable  
ret = ma_return_fonction()  
print(ret) # "coucou"
```

valeurs de retour multiples: unpacking

```
def ma_fonction():  
    return "ret1", "ret2", "ret3"  
  
r1, r2, r3 = ma_fonction()
```

passages par référence

```
def plus_2(param):  
    print(id(param)) # même id que "p"  
    param += 2  
    print(id(param)) # changement de id car modif  
    return param  
  
p = 10  
print(id(p))  
ret = plus_2(p)  
print(id(ret)) # même id que "param" retourné
```

passages par référence: cas mutable

```
def my_append(lst, elem)
    lst.append(elem)

l = [1, 2, 3]
my_append(l, 4)
print(l) # [1,2,3,4] : passage par référence + fonction interne
```

portée: variables locale vs globale

```
def ma_fonction():  
    l = "WORLD" # variable locale à la fonction  
    print(f"{g} {l} !") # lecture de g globale  
    g = "HELLO" # UnboundLocalError !!!  
  
g = "hello" # variable globale à la fonction  
ma_fonction()  
print(l) # NameError: l n'existe pas en dehors de la fonction
```


portée: modification une globale

```
def ma_fonction():  
    global g # on peut modifier la valeur globale en tant que globale  
    g = g.upper()  
    print(g, id(g))  
  
g = "hello" # variable globale à la fonction  
print(g, id(g))  
ma_fonction()  
print(g, id(g))
```

paramètres: positionnels à la définition

```
def ma_fonction(p1, p2):  
    return p1, p2  
  
ma_fonction("p1", "p2"): # "p1", "p2" : même position !!!  
ma_fonction(), ma_fonction("p1") # ERREUR: paramètres obligatoires
```

paramètres: nommés à la définition

```
# p2 peut être omis à l'appel  
# et remplacé par une valeur par défaut  
def ma_fonction(p1, p2="default"):  
    return p1, p2  
  
ma_fonction("p1", "p2"): # "p1", "p2" : normal  
ma_fonction("p1") # "p1", "default"
```

“ *dans la définition: on écrit d'abord les paramètres positionnels, parce qu'ils sont positionnels (!!)* et ensuite les paramètres nommés ”

paramètres: appel nommés

```
def ma_fonction(p1, p2):  
    return p1, p2  
  
# flécher les valeurs sur les params  
# on appelle les paramètres dans l'ordre qu'on veut  
ma_fonction(p2="p1", p1="p2") # "p2", "p1"
```

“ *intéressant quand une fonction a beaucoup de paramètres !!!* ”

attention sur les appels

```
def func(a, b="default", c=3.14):  
    pass
```

```
func(1, 8.12)
```



cet appel par défaut
concerne b !!!

erreur d'appels

```
func(1, b="qqch", 8.12)
```



ERREUR:

pas d'appel positionné
après un appel nommé

"variadics" positionnels à la définition

```
## permet d'autant de paramètres positionnels à l'appel qu'on veut  
def ma_fonction(*args):  
    # args est un tuple  
    return args  
  
ma_fonction("p1", "p2", "p3") # ("p1", "p2", "p3")
```

“ *exemple emblématique: print(*values)* ”

"variadics" positionnels à l'appel

```
def ma_fonction(p1, p2):  
    return p1, p2
```

```
params = ["p1", "p2"]
```

```
# les éléments de params sont dispatchés
```

```
# en tant que paramètres dans la signature de la fonction
```

```
ma_fonction(*params)
```


"variadics" nommés à la définition

```
## permet d'autant de paramètres nommés à l'appel qu'on veut
def ma_fonction(**kwargs):
    # kwargs est un dict
    return kwargs

ma_fonction(k1="v1", k2="v2", k3="v3")
# {"k1": "v1", "k2": "v2", "k3": "v3"}
```

“ *utilisés pour des options secondaires, cosmétiques ou métadonnées, en-têtes* ”

"variadics" nommés à l'appel

```
def ma_fonction(p1, p2):  
    return p1, p2  
  
params = {"p1": "v1", "p2": "v2"}  
# les valeurs de params sont dispatchées  
# en tant que paramètres dans la signature de la fonction  
# selon les clés de params  
ma_fonction(**params) # "v1", "v2"
```

restreindre les appels de paramètres

```
from inspect import signature # module d'introspection
def ma_fonction(a, /, b, *, c):
    pass

for param in signature(ma_fonction).parameters.values():
    print(param.name, param.kind)
# a POSITIONAL_ONLY : appel positionnel uniquement
# b POSITIONAL_OR_KEYWORD: appel quelconque
# c KEYWORD_ONLY: appel nommé uniquement
```

- “ *"/" appeler les paramètres principaux / obligatoires, de façon positionnel, à gauche*
- "*": appeler les nombreux paramètres de la fonction pour en comprendre l'usage grâce aux noms de paramètres, à droite*

signature complète

```
def func(pos1, pos2, ... , /, pos_opt, ..., *[args], opt1=..., opt2=..., **kwargs):  
    pass
```

paramètres positionnels appelables uniquement de façon positionnelle

paramètres positionnels appelables de façon positionnelle OU nommée

paramètres nommés appelables uniquement de façon nommée

signature universelle

```
def func(*args, **kwargs)
```



on peut appeler cette fonction de n'importe quelle façon !!!
très utile pour les DECORATEURS (notion avancée ...)

documenter, annoter une fonction

```
def ma_fonction(a: int, b: float=3.14) -> float:
    """
    DOCSTRING EN PREMIERE LIGNE !!!
    ma_fonction: retourne la puissance d'un réel
    a: int: exposant
    b: float (pi par défaut)
    """
    return b ** a

help(ma_fonction) # hover tooltip dans l'IDE
print(ma_fonction.__doc__, ma_fonction.__annotations__)
```

“ **ATTENTION** les annotations ne sont qu'informatives,
il n'y a aucun contrôle de type a priori en python !!!

fonctions lambda

- Fonctions **à usage unique, sans nom, d'une seule expression**
- Les lambda sont affectables *!= Pythonique*

```
y = 4 # variable libre  
l_func = lambda x, z, *args: x + y + z # x, z variables liées  
l_func(1, z=3)
```

usages pythoniques des lambda

- paradigme de *programmation fonctionnelle* et *lambda calcul*
- Façon de structurer le code en *composant des fonctions*

```
from functools import reduce
numbers = [5.2, 88, -3.14, 10]
# fonction MAP: transformer un vecteur par une fonction (RAPIDE)
squares = list(map(lambda x: x**2, numbers))
# fonction FILTER: filtrer //
positive = list(filter(lambda x: x >= 0, numbers))
# fonction SORTED: trier selon le retour de la lambda
abs_sort = sorted(numbers, key=lambda x: abs(x))
# fonction REDUCE: réduire la dimension d'un vecteur,
# ici vecteur (dim 4) => scalaire (dim 0)
somme = reduce(lambda acc, new_nb: acc + new_nb, numbers)
```


V. modules & packages

modules

- Un **module** est un fichier source, suffixé par « **.py** »
 - le mot clé **import** injecte le contenu d'un module dans un autre.
 - L'**espace de nom** du module importé permet de reconnaître et d'utiliser ce contenu
- “ 1. *un module contenant seulement des fonctions est une bibliothèque*
2. *Module importé => exécuté donc attention aux affichages !!!* ”

import: schéma

```
# imported_module.py
```

```
my_var = "truc"
```

```
def my_func():  
    return my_var
```

```
## WARNING !!  
print("coucou")
```

Exécute

```
# calling_module.py
```

```
import imported_module
```

```
print(imported_module.my_var)
```

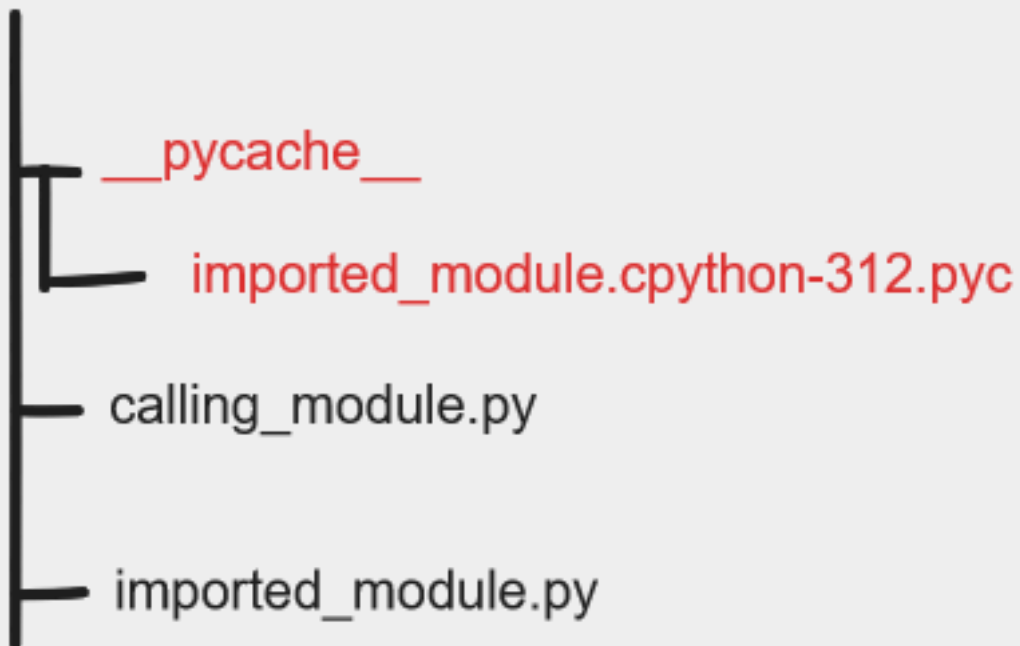
```
print(imported_module.my_func())  
    espace de nom
```

imports sans espace de nom

- `from imported_module import my_func`
 - permet un import **sans espace nom**
 - *MAIS tout le module importé est exécuté !!!*
 - *Attention* aux *collisions de noms* avec d'autres variables importées ou locales

```
from imported_module import my_var, my_func
# tout le contenu sans espace de nom
from imported_module import *
# en cas de collision de noms
from imported_module import my_func as alias_func
```

le dossier "__pycache__"



- 1ère exécution du module `calling_module.py`
 - + exécution de `imported_module.py`
 - + génération du cache pseudo-compilé
- 2ème exécution du module `calling_module.py`
 - + exécution de `imported_module.cpython-312.pyc`
 - + bcp plus RAPIDE !!!!

introspection des modules

- utilisation d'une *docstring* `""" doc """` en première ligne du module
- utilisation de `dir(imported_name)`
 - *un module est un objet python comme un autre !!!*
- en particulier l'attribut `__name__`
 - contient le *nom du fichier importé* sans extension

BEST PRACTICE: bloc "programme principal"

- on peut utiliser l'attribut `__name__` sur le *module appelant*
- de fait quand un module est exécuté directement par l'interpréteur
 - `__name__` vaut `"__main__"`

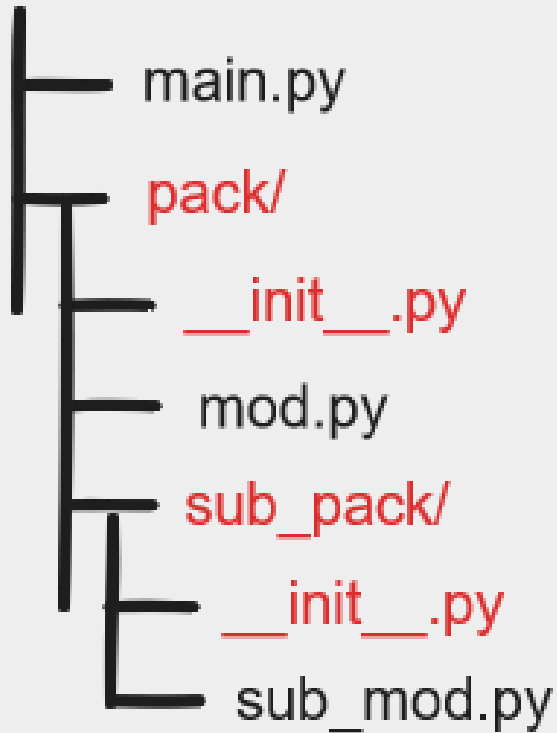
```
# module calling_module.py
import imported_module
...
if __name__ == "__main__":
    # CE BLOC NE SERA EXECUTE QUE SI LE MODULE COURANT EST EXECUTE
    # DIRECTEMENT PAR L'INTERPRETEUR !!!
```

les packages

- Un package est un répertoire contenant des modules et un fichier « **__init__.py** »
- un package peut se décomposer en *sous package*
- pour importer un module à partir d'un package on va utiliser
 - `<package>[.<subpackage>].<module>`; *chemin python*

```
# chemins d'imports absolus, depuis le module main.py
import pack.mod
from pack.mod import my_func
from pack.sub_pack.sub_mod import sub_func
```

package: import absolus



si l'on veut importer une fonction de sub_mod dans mod

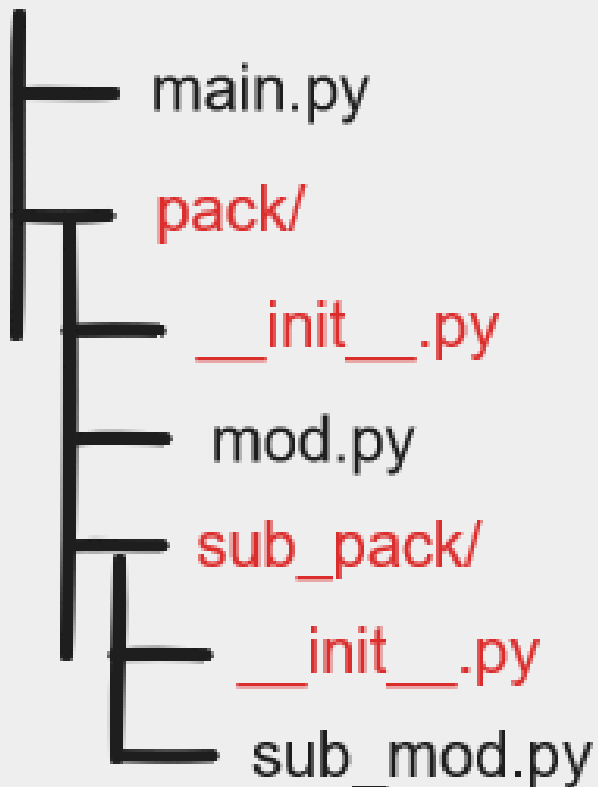
```
from pack.sub_pack.sub_mod import sub_func
```

i.e : les chemins intermédiaires sont calculés

- depuis le module main.py qui est le programme principal

on utilise pas un module d'un package comme programme principal

package: import relatif



si l'on veut importer une fonction de sub_mod dans mod

```
from .sub_pack.sub_mod import my_sub_func
```

BAD PRACTICE

si l'on veut importer une fonction de mod dans sub_mod

```
from ..mod import my_func
```

ici l'import est calculé à partir d'un module du package vers un autre module du même package parent

La Bibliothèque standard

- contient les modules *installés avec python*
- modules variés qui justifient l'adage: **python sert à tout !**

module	Desc.
re	expression régulières
math	Fonctions mathématiques
datetime	Gestion des dates et heures
locale	Internationalisation (gettext)
sys, os, io	Intéraction avec l'OS
csv, xml, json	Formats de fichiers
zlib, zipfile	Formats de compression
tkinter	Interfaces graphiques

Exemple: datetime

- manipule les dates et les heures

```
from datetime import datetime  
  
# annee, mois et jour sont obligatoires  
datetime(annee, mois, jour, heure, minute, seconde, microseconde, fuseau_horaire)
```

“ *la signature canonique n'est pas bcp utilisée !* ”

datetime: création

```
from datetime import datetime

# à partir d'une chaîne de caractères (strptime: "p" pour parse)
dt = datetime.strptime("2021-12-17 10:14:20", "%Y-%m-%d %H:%M:%S")

# maintenant
now = datetime.now()

# nb de secondes depuis le 1er janvier 1970, dit temps "Unix"
dt = datetime.fromtimestamp(1639732460.0)
```

datetime: usages

```
from datetime import datetime

dt = datetime.now()
dt.year, dt.month, dt.hour, dt.date(), dt.time()

# 0 => lundi, 6 => dimanche
dt.weekday()

# timestamp: nb de secondes depuis 1er janv. 1970
dt.timestamp()

# formater la date en chaine (strftime: "f" pour format)
dt.strftime("%Y/%m/%d")
```

Datetime : durées (timedelta)

```
# arithmétique de date
premier_confinement = datetime.strptime("14/03/2020", "%d/%m/%Y")
aujourd'hui = datetime.now()

duree_covid = aujourd'hui - premier_confinement

# objet timedelta
duree_covid.days, type(duree_covid)

cuisson_oeuf_coque = timedelta(minutes=3)
atable = datetime.now() + cuisson_oeuf_coque
atable.time()
```

Exemple: re

- Les expressions régulières sont des chaines de caractères
- qui décrivent des *modèles de chaines de caractères*
- Via une syntaxe particulière
- <https://cheatography.com/davechild/cheat-sheets/regular-expressions/>

re: exemple

```
import re

# est ce que le modèle matche la chaine depuis le début ?
m = re.match(
    "(\w+) weighs (\d+(\.\d+)?) (kg|g|lbs)",
    "peter weighs 144 lbs")

# les parenthèses capturant permet d'utiliser des portions de la regex
m.groups(), m.group(1)

# est ce que la chaine contient un float positif ?
re.search("(\d+(\.\d+)?)", "peter weighs 144.5 lbs")

# chercher et remplacer
re.sub("(\d+(\.\d+)?)", "*****", "peter weighs 144.5 lbs")
```


VI. Programmation Orientée Objet

paradigmes de programmation

- Modèles d'écriture de code comme chemin du pb vers la solution
- **Programmation impérative**: bloc d'instructions => *machine d'état*
- **Programmation fonctionnelle**: fonctions composées => $S = f \circ g(P)$
- « **POO** »: classes et d'objets => *assemblage de « légos »*

classes et objets

- 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 le produit fabriqué à partir de son plan
- “ *le type d'un objet est sa classe* ”

attributs, instance

```
class BankAccount:  
    # attributs: variables internes  
    _id = 0  
    balance = 0.  
  
# création de la variable acc de type BankAccount  
# instantiation de l'objet "acc" de classe BankAccount  
acc = BankAccount()  
# manipulation des attributs hérités de la classe  
acc._id = 21345345  
print(acc._id)
```

“ l'objet est une **instance** de la classe

”

méthodes

```
class BankAccount:  
    _id = 0  
    balance = 0.  
    # méthodes: fonctions internes  
    def is_overdraft(self):  
        return self.balance < 0  
  
acc = BankAccount()  
acc.balance = 100  
print(acc.is_overdraft()) # acc est "self" donc pas de paramètre !!
```

- “ **self** est automatiquement le premier paramètre d'une méthode
self représente l'objet instancié ici **acc**
donc le "paramètre **self**" n'est pas *nourri à l'appel*

attributs / méthodes - publics / privés

- a priori on peut manipuler les attributs d'un objet depuis l'extérieur de la classe => **attributs publics**
- un attribut préfixé par `__` ne peut plus être lu ou écrit depuis l'extérieur => **attribut privé** but: *structurer, formater, nettoyer*

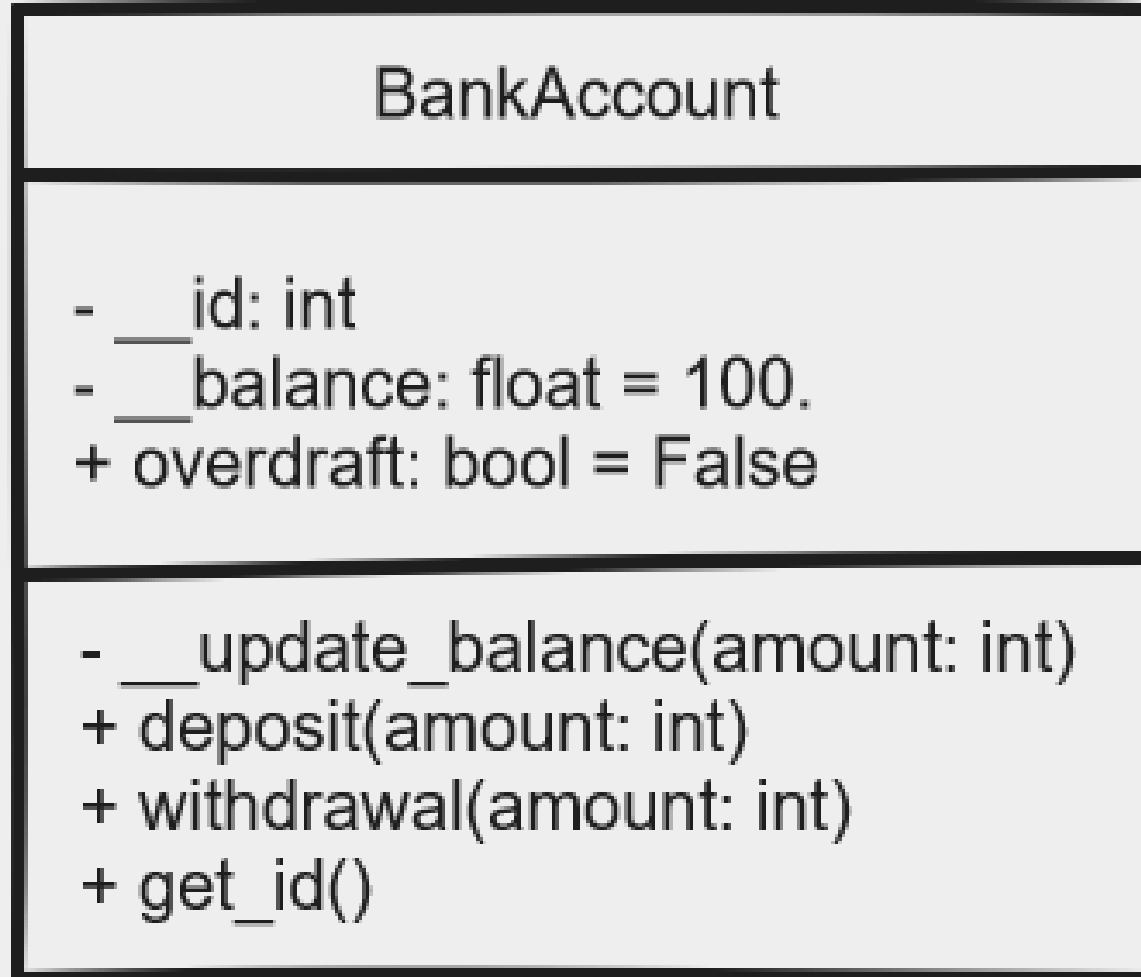
```
class BankAccount:  
    __id = 0  
    # méthodes publiques  
    def get_id(self): return self.__id # lecture depuis la classe !!  
    def set_id(self, _id): self.__id = _id # écriture depuis la classe !!  
  
acc = BankAccount()  
acc.set_id(3434555) # setter: méthode publique pour écrire un attribut  
print(acc.get_id()) # getter: méthode publique pour lire un attribut
```

attributs de classe et d'objet

```
class BankAccount:  
    # attribut de classe  
    balance = 0.  
    def get_balance(self): return self.balance # attribut d'objet  
    def set_balance(self, balance): self.balance = balance  
  
acc = BankAccount()  
acc.set_balance(100)  
print(acc.get_balance()) # 100  
print(BankAccount.balance) # 0.
```

- “ *au moment de l'instanciation les attributs d'objet **héritent** *des attributs de classe*
Par la suite, les attributs d'objets deviennent **indépendants**

modélisation UML d'une classe



NOM

- Attributs Privés

+ Attributs publics

- Méthodes privées

+ Méthodes publiques

attributs/méthodes "magiques"

- attributs/méthodes *préfixés et suffixés* par `__`
- gèrent des *comportement* des objets instanciés

```
class BankAccount:  
    # génère et renseigne des attributs d'objet dès l'instanciation  
    def __init__(self, _id: int, balance: 100.):  
        self.__id = _id  
        self.__balance = balance  
    # donne un sens à l'affichage et la conversion en str de l'objet  
    def __str__(self) -> str:  
        return f"id: {self.__id} balance: {self.__balance}"  
  
acc = BankAccount(34554) # exécute la méthode __init__  
print(acc), str(acc) # exécute __str__
```

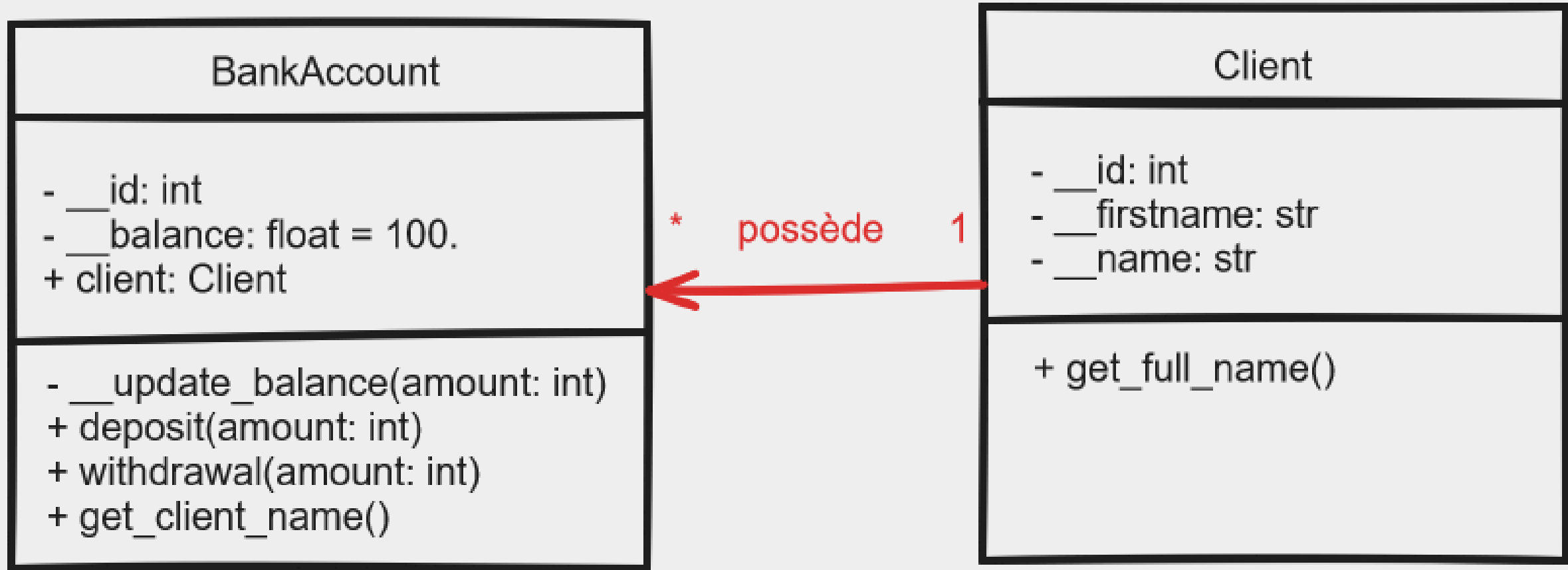

injection de dépendances

```
class Client:
    def __init__(self, firstname: str, name: str):
        self.__firstname = firstname; self.__name = name
    def get_full_name(self):
        return f"{self.__firstname.capitalize()} {self.__name.upper()}"

class BankAccount:
    def __init__(self, balance: float, owner: Client):
        self.__balance = balance; self.__owner = owner
    def get_client_name(self): return self.__owner.get_full_name()
```

“ *Injection de dépendance: association faible entre objets en utilisant qu' **uniquement l'interface publique*** ”

modélisation UML de l'association



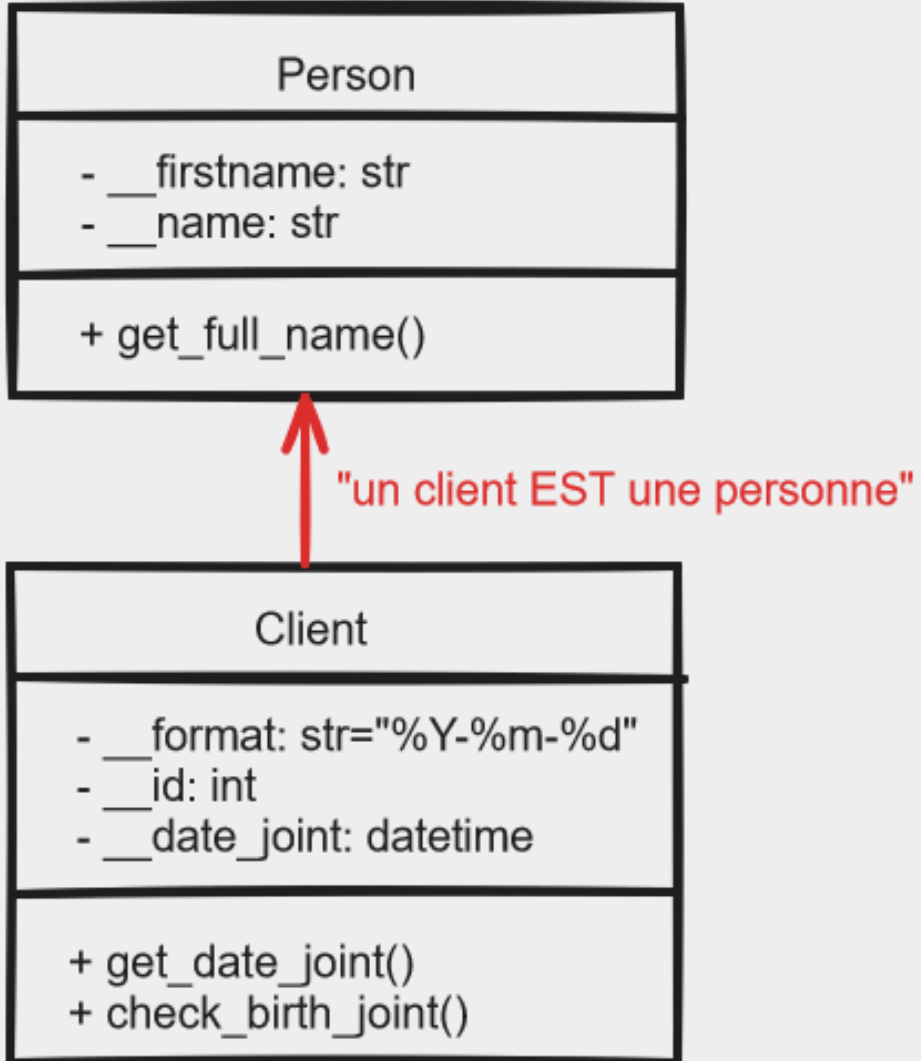
héritage

```
class Person:
    def __init__(self, f, n):
        self.__f = f
        self.__n = n
    def get_full_name(self): return f"{self.__f.capitalize()} {self.__n.upper()}"

class Client(Person):
    def __init__(self, _id, f, n):
        self.__id = _id
        # super(): appel à l'initialiseur de la classe mère (Person)
        # fournit les attributs d'objet de l'objet courant (cl)
        super().__init__(f, n)

cl = Client(34534, "bob", "smith")
cl.get_full_name() # méthode héritée
```

modélisation UML de l'héritage



notion d'interface

- ensemble des **noms & signatures des méthodes** à utiliser
- on utilise le module **abc** et la classe **ABC**
- pour faire hériter une classe qui va définir une interface
- une telle classe est dite **abstraite**

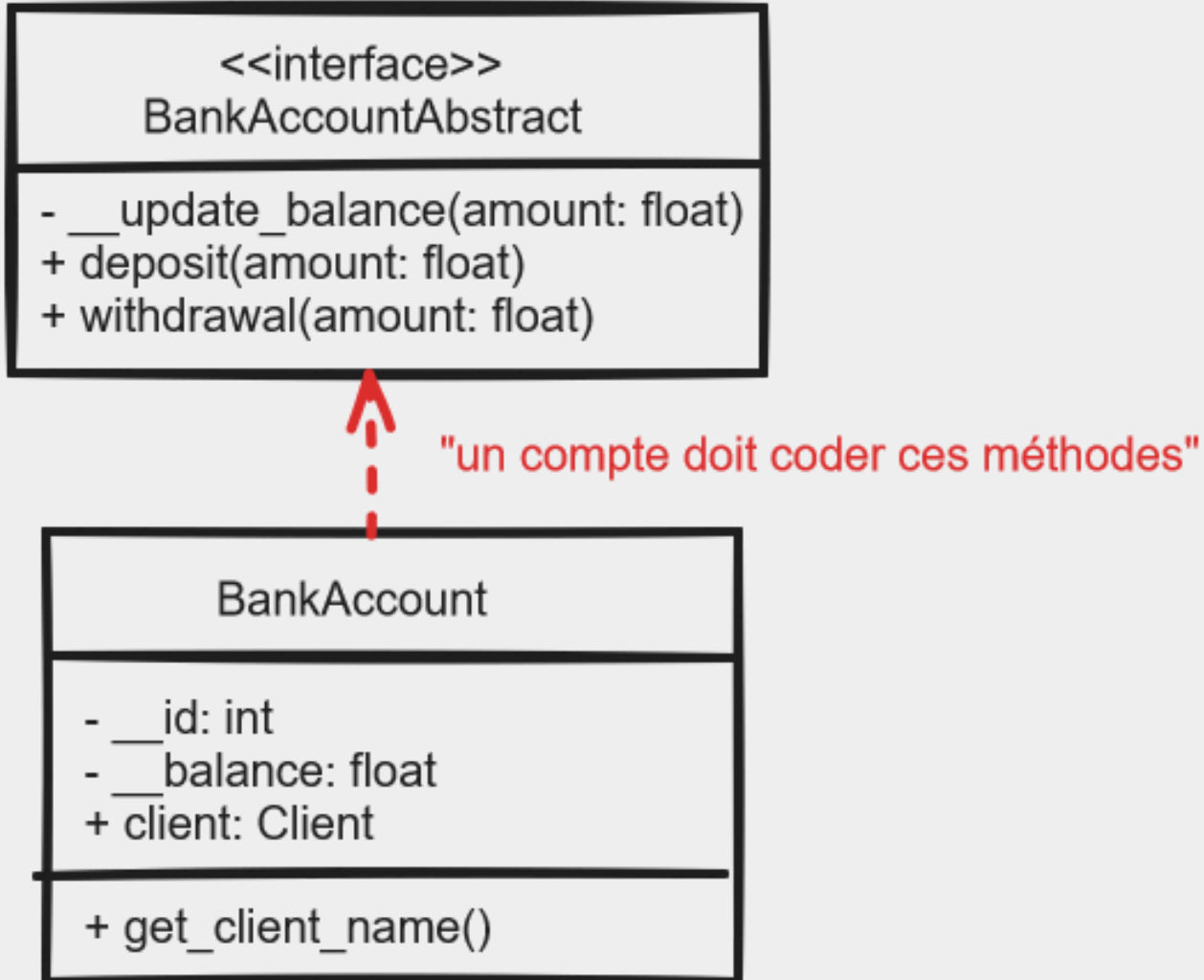
“ **AB** *stract* **C** *lass*

”

exemple de classe abstraite

```
# cette classe ne peut pas instancier des objets  
# toute classe héritant de cette classe DOIT implémenter ces méthodes  
class BankAccountAbstract(ABC):  
    def __init__(self, _id, balance): pass  
    def get_id(self): pass  
    def get_balance(self): pass  
  
class BankAccount(BankAccountAbstract):  
    def __init__(self, _id: int, balance: float=100.):  
        self.__id = _id; self.__balance = balance  
    def get_id(self): return self.__id  
    def get_balance(self): return self.__balance
```

modélisation d'une interface



en python, tout est objet !!!

```
class Truc:  
    pass  
  
t = Truc()  
type(t), isinstance(t, Truc) # Truc est un type de donnée  
  
m = "machin"  
type(m), isinstance(m, str) # str est une classe !!!  
  
# fonction de conversion signifie en fait INSTANCIATION  
# on crée l'objet _str de classe str  
# en exécutant la méthode __init__ de str !!!  
_str = str(1)
```