Septembre 2015

© ADOK Gilles Vanderstraeten
gillesvds@adok.info

Historique

- Créé par Guido van Rossum (NL) en 1991 (clin d'oeil aux Monty Python).
- Version 2 en 2000.
- Version 3 en 2008 incompatible avec la version 2.
- Actuellement en version 3.5.

Caractéristiques générales

- Open-source et gratuit.
- Interprété sans typage statique.
- Langage objet dès sa création.
- Utilisable en POO ou/et en mode procédural.
- Présente encore beaucoup d'incohérences même dans sa dernière version.
- Eclipse : plug-in PyDev.

Domaines d'utilisation

- Initiation à la programmation car proche du pseudo-code.
- Traitement de (petites) structures de données.
- **Traitements mathématiques (Bibliothèque Sage).**
- **Langage des développeurs 3D dans Max, Maya, Cinema 4D...**
- **Interfaces, "glue" entre applicatifs.**
- Applications web avec les frameworks Django et TurboGears.
- Applications desktop avec Qt (possible mais pénible).

Documentation

- //www.python.org : site officiel (anglais).
- //www.python.org/dev/peps : *Python Enhancement Proposals* (anglais).
- //docs.python.org : doc officielle (anglais).
- //docs.python.org/3/glossary.html : glossaire (anglais)
- //www.python-forum.org : forum (anglais).
- //www.afpy.org : association francophone pour la promotion de Python.

Installation sous Windows

- Binaire disponible gratuitement sur le site officiel.
- Ajoute le chemin de l'exécutable dans le PATH.
- Fonctionne immédiatement via l'invite de commandes de Windows.
- `> python` pour vérifier la version (`exit()` pour quitter).

Premières lignes de code

- La commande `python` lance un compilateur interactif.
  Exemple :

```
> python
Python 3.3.2...
>>> a='Toto'
>>> b='Lulu'
>>> print(a+' aime '+b)
Toto aime Lulu
```

Premier programme

- Créer le fichier `hello.py` contenant la ligne :

```
print('Hello !')
```

- Lancer son exécution dans la console DOS :

```
python hello.py
```

Sensibilité à la casse

- Python est intégralement sensible à la casse (mots-clés, identifiants...).

Fin de ligne

- Le caractère de fin de ligne LF (Linux) ou CRLF (Windows) termine chaque ligne du code-source.
- Les expressions entre `()`, `[]` ou `{}` peuvent s'étendre sur plusieurs lignes.
- Le `;` permet de séparer plusieurs instructions sur une même ligne.

Indentation obligatoire

 Une indentation correcte est obligatoire pour fixer l'étendue des blocs.

Commentaires

- Commentaires d'une ligne avec #.
- Pas de commentaires multilignes. Les `...` ne fonctionnent pas.

Sortie

`print(objet,sep=' ',end='\n')`

- Fonction *top-level*.
- Ajoute un saut de ligne par défaut.
- Exemples :

```
print('toto')                          # toto
print('toto','lulu')                   # toto lulu
print('toto','lulu',sep=':',end=';') # toto:lulu;
```

 Les arguments nommés (*keyword arguments*) seront détaillés plus loin.

 La méthode `printf()` est obsolète. Voir plus loin `format()`.

Caractères spéciaux utiles

- Saut de ligne : `\n`
- Tabulation : `\t`

Entrée

Récupérer les arguments de la ligne de commande

- Exemple avec la variable automatique `argv` du module `sys` :

```
import sys
print(sys.argv[2])

> test.py toto lulu
lulu
```

- Retourne une liste indexée des arguments (chaînes).
- Comme en C, chemin du script en premier argument.


L'importation de modules sera détaillée plus loin.

Entrée

`input(`*`prompt`*`)`

- Fonction *top-level.*
- Affiche le *prompt* si présent puis retourne la chaîne saisie au clavier.
- Entrée toujours considérée comme une chaîne.
- Supprime le retour à la ligne.
- Exemples :

```
prenom=input("Prenom ? ")

n=int(input("n ? "))
```

Terminer

`exit(`*`message`*`)`

- Fonction *top-level.*
- Arrête l'exécution.
- Affiche le *message*.

Fonctions *top-level* ???

- Elles sont regroupées dans le module `builtins` (voir plus loin).
- Elles ne sont pas des méthodes de `object`, mère de tous les objets.
- Elles sont disponibles en mode procédural.
- Elles sont parfois des alias (en mode procédural) de méhodes contenues dans des classes spécifiques MAIS PAS TOUJOURS.
- Elles peuvent être appelées directement sans référence à une classe.
- Exemples :
  - `int()` est le constructeur du type (la classe) `int`.
  - `hex()` est un alias de `int.hex()` MAIS PAS de `float.hex()`.
  - `abs()` N'EST PAS un alias, elle est différente de `math.fabs()`.
  - `format()` est un alias de `str.format()`.

Mots-clés

- False, True, None
- from, import
- def, global, nonlocal, return, yield, lambda
- and, or, not
- if, elif, else
- for, in, while, break, continue
- try, except, finally, raise
- class, is
- with, as
- del, pass, assert

Littéraux numériques

- Exemples d'entiers :

```
123         (décimal)
0b10010011  (binaire)
0o75        (octal)
0xE         (hexadécimal)
```

- Exemples de réels :

```
123.0
.123
1.23e2
```

- Exemple de complexe (avec partie réelle nulle) : `3j`.
- Exemple d'un littéral complexe avec partie réelle non nulle : `4+3j`.

Les signes `-` et `+` ne font pas partie des littéraux, ce sont des opérateurs unaires.

Littéraux chaînes

- Indifféremment entre simples-quotes ou doubles-quotes.
- Les triples simples-quotes ou doubles-quotes permettent d'étendre la chaîne sur plusieurs lignes.
- Le caractère d'échappement est `\`.

Concaténation implicite

- Des littéraux séparés par un ou plusieurs espaces sont implicitement concaténés. Exemple :

```
print('toto' 'aime'   'lulu') # 'totoaimelulu'
```

Types numériques

Module `numbers`

- `Complex > Real > Rational > Integral  > Boolean`

 Ces classes sont abstraites, elles ne peuvent être instanciées.

 Les constantes `False` et `True` valent en réalité respectivement `0` et `1`.

Module `decimal`

- Permet les calculs décimaux exacts.

Module `rational`

- Permet les calculs rationels exacts.

 La notion de "module" sera détaillée plus loin.

Types numériques

Dans la pratique...

- Le type `int` (en réalité, une classe) implémente les méthodes de la classe abstraite Integral (et ajoute même une méthode).
- Même chose avec les types (classes) `float` et `bool`.
- Le traitement des `int` et `float` est dépendant de la plateforme mais la longueur des `int` est illimitée.

Opérateurs arithmétiques

`+`	Addition, ex : `1+2` Opérateur unaire, ex : `+3`
`-`	Soustraction, ex : `3-1` Opérateur unaire, ex : `-3`
`*`	Multiplication
`/`	Division réelle
`//`	Division entière
`%`	Modulo
`**`	Puissance

Fonctions *top-level* arithmétiques

abs(n)	Valeur absolue de **n**
max(...)	Maximum d'une séquence
min(...)	Minimum d'une séquence
pow(x,y,[z])	X puissance Y, le tout modulo Z
round(x,n)	Arrondi de X à n décimales (alias de `math.round`)

Module `math`

- Il contient des constantes et des méthodes statiques. Exemples :

math.pi	Constante Pi.
math.ceil(x)	Arrondi supérieur de X.
math.floor(x)	Arrondi inférieur de X.
math.trunc(x)	Partie entière de x.
math.round(x[,n])	Arrondi de X à n décimales.
math.factorial(x)	Factorielle (illimitée) de X.
math.log(x,b)	Log de X en base b (e par défaut)
math.sin(x)	Sinus de X (radians).

TD : `factorial`

- Pour tester la capacité de Python à manipuler des nombres entiers illimités, calculez la factorielle de `123456` voire d'avantage si votre PC est rapide.

Solution : factorial.py

- Toujours à l'aide du module `math`, trouvez combien il y a de chiffres dans le résultat (et appréciez la rapidité !).

Solution : factorial2.py

Conversions entre types numériques

Conversions implicites

- Exemples :

```
2+3.0              # 5.0
10*False+100*True # 100
```

Conversions explicites

- Exemples avec les fonctions *top-level* :

```
int(5.9) # 5
float(5) # 5.0
bool(5)  # True
bool(0)  # False
bool(-1) # True
```

Evaluation booléenne

- Les valeurs évaluées à faux sont :
  - None (constante "rien", comparable et affectable)
  - `False`
  - `0` (ou `0.0`)
  - `''` (chaîne vide)
  - `()` (tuplet vide, voir plus loin)
  - `[]` (liste vide, voir plus loin)
  - `{}` (jeu vide, voir plus loin)
- Toutes les autres valeurs sont évaluées comme vraies.

Type chaîne (classe `str`)

 L'ancienne classe `String` est obsolète.

- La nouvelle classe `str` offre les méthodes habituelles.
- Documentation : //docs.python.org/3/library/stdtypes.html#text-sequence-type-str
- Exemples :
  - `capitalize()`
  - `find(`*`sub`* `[,`*`start`*`] [,`*`end`*`])`
  - `lower()`
  - `replace(`*`old`*`, `*`new`* `[,`*`count`*`])`
  - `split(`*`sep`*`=None, `*`maxsplit`*`=-1)`
  - `upper()`

Opérateurs sur chaînes

+	Concaténation explicite
*****	Répétition

Fonctions *top-level* sur chaînes

len(s)	Nombre de caractères de S

`format()` et `str.format()`

- Documentation :
  //docs.python.org/3/library/functions.html#format
  //docs.python.org/3/library/string.html#format-string-syntax
- Pour les habitués de `printf()`, il suffit souvent d'ajouter des `{}` et de remplacer `%` par `:`.
- La fonction `format()` ne permet de formater qu'une valeur. Exemple :

```
format(12,'.2f') # 12.00
```

- La méthode `str.format()` permet d'avantage. Exemple :

```
"{} oeufs = {:.2f} EUR".format(6,3)
# 6 oeufs = 3.00 EUR
```

Extraits de chaînes

- Extraction d'un caractère comme dans un tableau indicé. Exemple :

```
"Toto aime Lulu"[10] # L
"Toto aime Lulu"[-4] # L (indexation négative)
```

- Extraction d'une tranche (*slice*) avec `[:]`. Exemple :

```
"Toto aime Lulu"[5:8] # aim (8 exclus)
"Toto aime Lulu"[5:]  # aime Lulu
"Toto aime Lulu"[:8]  # Toto aim
"Toto aime Lulu"[:]   # Toto aime Lulu
```

Conversions entre types numériques et chaînes

Conversions implicites

 Aucune conversion implicite.

Conversions explicites

- Exemples avec les fonctions *top-level* :

```
str(5.9)       # '5.9'
float('5.9e3') # 5900.0
float('5')     # 5.0
int('5')       # 5
int('5.9')     # Erreur, réel !
bool('toto')   # True
bool('')       # False
bool('0')      # True
```

TD : `factorial` (suite)

- Trouvez le nombre de chiffres sans utiliser le module `math`.

Solution : factorial3.py

Identifiants

- Pour tous les identifiants (variables, fonctions, classes...) :
  - Peuvent contenir `a-z`, `A-Z`, `0-9` et `_`.
  - Ne doivent pas commencer par un chiffre.
  - Longueur infinie.

Variables

- Pas de déclaration donc initialisation obligatoire.
- Typage dynamique.
- Exemple :

```
a=3
a='toto'
a=3.5
print(a) # 3.5
```

Constantes

- Il n'existe pas de constantes en Python.

Opérateurs d'affectation

=	Affectation simple, ex : `a=3` Affectation multiple (*parallel assignment*), ex : `a,b=1,2` Affectation chaînées, ex : `a=b=3`
+=	Affectation et addition (ou concaténation) combinées
-=	Affectation et soustraction combinées
***=**	Affectation et multiplication (ou répétition) combinées
/=	Affectation et division réelle combinées
//=	Affectation et division entière combinées
%=	Affectation et modulo combinées
****=**	Affectation et puissance combinées

TD : opérateurs

NIR 13

- Ecrire un programme qui lit un numéro de sécurité sociale (NIR13 à 13 chiffres, sans la clé) puis affiche la clé correspondante (sur 2 digits).
- INFO
  La clé est le résultat de la soustraction à 97 du reste de la division du numéro par 97.

Solution : nir13.py

Types mutables et immutables

- RAPPEL :
  Une variable d'un type **immutable** ne peut recevoir qu'une valeur unique dans son cycle de vie, à la façon d'une constante. Cependant, contrairement à une constante, une nouvelle affectation ne génère pas une erreur mais crée automatiquement une nouvelle instance.
- **Tous les types numériques, ainsi que le type `str` sont immutables.**
- Exemple (la fonction *top-level* `id()` retourne l'adresse mémoire) :

```
s="toto"
print(id(s))  # 44433792 (adresse mémoire de "s")
s="lulu"
print(id(s))  # 36036424 (nouvelle adresse)
s+='!';
print(id(s))  # 44522048 (nouvelle adresse)
s[1]='i'      # Erreur, immutable !
```

Opérateurs de comparaison

`==`	Egalité simple
`!=`	Non-égalité simple
`is`	Egalité totale (même adresse mémoire)
`is not`	Non-égalité totale
`<`	Strictement inférieur
`>`	Strictement supérieur
`<=`	Inférieur ou égal
`>=`	Supérieur ou égal

Opérateurs logiques

`not`	Négation
`and`	ET
`or`	OU (inclusif)

TP : opérateurs logiques

- Complétez par `true` ou `false` le tableau suivant.

A	B	A and B	A or B	!(A and B)	!A or !B
true	true				
true	false				
false	false				

- Que remarquez-vous ?

Tests conditionnels

`if... elif... else...`

- Syntaxe :

`if` *exp1*`:`
  ...code excécuté si *exp1* vraie...
`elif` *exp2*`:`
  ...code excécuté si *exp1* fausse mais *exp2* vraie...
`else:`
  ...code exécuté si *exp1* et *exp2* fausses...

- Un et un seul `if`.
- Zéro, un ou plusieurs `elif`.
- Zéro ou un `else`.

Attention à l'indentation obligatoire !

Test conditionnel rapide

... if... else...

- Exemple :

```
signe='positif' if valeur>=0 else 'négatif'
```

TD : tests conditionnels

NIR 15

- Dupliquer le programme `nir13.py`. Le modifier pour qu'il réalise les opérations suivantes :
  - Demander la saisie du numéro complet au format : SAAMMDDNNNRRR-CC.
  - Vérifie la structure du numéro (hors cas spéciaux...) :
    - S (sexe) est 1 ou 2.
    - MM (mois) est compris entre 01 et 12 (sauf si inconnu).
    - DD (dpt.) est compris entre 01 et 95 ou 99 (étrangers).
    - Présence du tiret.
    - CC (clé) doit correspondre au numéro.
  - Indique si le numéro est valide ou quelle partie est incorrecte.

 Lorsqu'elles doivent être numériques, les données sont supposées l'être.

Solution : nir15.py

Séquences : tuples (classe `tuple`)

- **Séries immutables, ordonnées de données hétérogènes.**
- Parenthèses recommandées (obligatoires dans certains cas).
- Immutables... mais peuvent contenir des données mutables.
- Exemples :

```
t=1,'toto',2.5     # packing
print(t)           # (1, 'toto', 2.5)
a,b,c=t            # unpacking
print(a,b,c)       # 1 toto 2.5
print(t[1])        # toto
print(t[1:2])      # ('toto',) slicing
t+=3,4,5           # append
t[1]='lulu'        # Erreur, immutable !
t=1,(3,'lulu',4),5 # Imbrication (et packing)
t=()               # Tuple vide
t=('unique',)      # Virgule obligatoire !
```

- NOTE : affectation mutiple (*parallel assignment*) = *packing-unpacking*.

Séquences : listes (classe `list`)

- **Séries mutables, ordonnées de données hétérogènes.**
- Traditionnellement utilisées pour des données homogènes.
- Exemples :

```
liste=[1,2,3]
print(liste)         # [1, 2, 3]
a,b,c=liste          # unpacking
print(a,b,c)         # 1 2 3
print(liste[1])      # 2
print(liste[1:2])    # [2] (slicing)
liste+=[4,5,6]       # append
liste[1]=9           # Ok, mutable
liste[1:3]=[7,8]     # Ok, mutable
liste=[4,[1,2,3],5] # Imbrication
liste=[]             # Liste vide
```

Séquences : *ranges* (classe `range`)

- **Séries immutables de nombres.**
- Essentiellement utilisées en conjonction avec le constructeur `list()`.
- Egalement utilisées pour les boucles indexées (voir plus loin).
- Exemples :

```
list(range(5))# [0, 1, 2, 3, 4]
list(range(2,5))# [2, 3, 4]
list(range(2,5,2))# [2, 4] (pas de 2)
```

- Sans `list()`, `range()` retourne un iterateur. Voir plus loin.
- La version `range(`*`start, stop, step=1`*`)` nécessite v3.3.
- La fonction *top-level* `range()` est un alias du constructeur de la classe.

Opérations communes aux séquences

str, tuple, list, range...

in	Présence d'une valeur scalaire (ou d'une séquence pour `str`)
not in	Absence d'une valeur scalaire (ou d'une séquence pour `str`)
+	Concaténation (sauf *ranges*)
*****	Répétition (sauf *ranges*)
[*i*]	Indexation de `0` à `len()-1` puis de `-1` à `-len()`.
[*a*:*b*]	*Slicing*
[*a*:*b*:*step*]	*Slicing* avec pas
len(*seq*)	Longueur (fonction *top-level*)
min(*seq*)	Minimum (fonction *top-level*)
max(*seq*)	Maximum (fonction *top-level*)

Dictionnaires (class `dict`)

- **Séries mutables, non-ordonnées de données hétérogènes indexées par des clés (table de hachage).**
- Exemples :

```
notes={'toto':12,'lulu':15}
print(notes)              # {'toto':12, 'lulu':15}
print(notes['toto'])      # 12
dico={}                   # Dictionnaire vide
print(list(notes.keys())) # ['lulu', 'toto']
```

Méthodes communes aux mutables

`list, dict...`

`.append(`*`x`*`)`	Ajoute d'un élément X à la fin de la séquence.
`.clear()`	Vide la séquence.
`.copy()`	Copie (équivalent à `[:]`)
`.extend(`*`seq`*`)`	Ajoute d'une séquence à la fin de la séquence.
`.insert(`*`i`*`,`*`x`*`)`	Insertion d'un élément X.
`.remove(`*`x`*`)`	Retire de la séquence le premier item égal à *x*.
`.reverse()`	Inverse l'ordre de la séquence.

Jeux (class set)

- **Séries immutables, non-ordonnées de données dédoublonnées.**
- Indexation, concaténation et imbrication non supportées.
- Exemples :

```
jeu={'toto','lulu','toto'} # Dédoublonnage auto.
print(jeu)                 # {'lulu', 'toto'}
a,b=jeu                    # unpacking
print(a,b)                 # lulu toto
print(len(jeu))            # 2 (fonction top-level)
j1=set('abracadabra')      # {'a', 'r', 'b', 'c', 'd'}
j2=set('tralala')          # {'a', 'r', 't', 'l'}
print(j1-j2)               # {'c', 'b', 'd'} différence
print(j1&j2)               # {'a', 'r'} intersection
print(j1|j2) # {'l', 'a', 'c', 'b', 'd', 'r', 't'} union
print(j1^j2) # {'l', 'c', 'b', 'd', 't'} diff. symétrique
```

 `set()` crée un jeu vide alors que `{}` crée un dictionnaire vide.

Parcourir str, tuple, list, set

for... in

- Exemple :

```
liste=[4,5,6]
for i in liste:
  print(i)
# 4
# 5
# 6
```

Parcourir `str, tuple, list, set` avec un index

`for... in enumerate()`

- `enumerate()` est une fonction *top-level*. Exemple :

```
liste=['toto','lulu','lili']
for i,prenom in enumerate(liste):
  print(i,prenom)
# 0 toto
# 1 lulu
# 2 lili
```

Parcourir `dict`

`for... in... .items()`

- Exemple :

```
notes={'toto':12,'lulu':15}
for nom,note in notes.items():
  print(nom+' : '+str(note)+'/20')
# lulu : 15/20
# toto : 12/20
```

Boucles indexées

for... in... range(...)

- Exemple :

```
liste=[4,5,6]
for i in range(0,len(liste)):
  print(liste[i])
# 4
# 5
# 6
```

Boucles while

- Exemple :

```
n=1
while n<4:
  print(n)
  n+=1
# 1
# 2
# 3
```

Boucles conditionnelles

`break`

- Provoque la sortie de la boucle en cours.

`continue`

- Provoque le saut à l'itération suivante.

`else`

- A la suite d'une boucle, sera toujours exécuté SAUF en cas de `break`.

 Pas d'étiquette de boucle en Python.

Compréhensions de listes (*list-comprehensions*)

- Nouvelle façon de créer des listes.
- Utilisées pour créer une liste en faisant subir un traitement aux éléments d'une autre.
- **S'applique de façon similaire aux dictionnaires et aux jeux.**
- Exemple :

```
liste=[4,5,6]
liste2=[i**2 for i in liste]
print(liste2) # [16, 25, 36]
```

Comparaison de séquences

str, tuple, list, range...

- Uniquement possible entre séquences de même type.
- Exemples d'assertions vraies :

```
(1,2,3)<(1,2,4)
[1,2,3]<[1,2,4]
'ABC'<'C'<'Lulu'<'Toto'
(1,2,3,4)<(1,2,4)
(1,2)<(1,2,-1)
(1,2,3)==(1.0,2.0,3.0)
```

TD : boucles 1

Fibonacci 300

- Ecrire un programme qui affiche les termes inférieurs à 300 de la suite de Fibonacci.

Solution : fibonacci300.py

Fibonacci 20

- Modifier le programme précédent pour qu'il affiche les 20 premiers termes de la suite (d'indice 1 à 20).

Solution : fibonacci20.py

- INFO
  La suite de Fibonacci est une série d'entiers naturels ainsi composée :
  - Le terme d'indice 0 est 0 (arbitrairement, jamais affiché).
  - Le terme dindice 1 est 1.
  - Le terme d'indice n vaut la somme des termes d'indice n-2 et n-1.

TD : boucles 2

Factorielle

- Ecrire un programme qui saisit une entrée puis affiche sa factorielle (sans utiliser la fonction prédéfinie du module `math`).

Solution : facto.py

 Python n'a pas de limite de taille pour les entiers. Testez votre CPU avec `n=100000` et soyez patient...

Factorielles

- Ecrire un programme qui affiche la factorielle des entiers de 1 à 10.

Solution avec 2 niveaux de boucles : facto10.py
Solution avec une seule boucle : facto10plus.py

- INFO
  La factorielle de zéro est 1 (arbitrairement).
  La factorielle d'un entier n strictement positif est : `n*(n-1)*(n-2)*...*1`.

TD : boucles 3

Verlan

- Sans utiliser une fonction d'inversion prédéfinie, écrire un programme qui saisit une chaîne puis affiche celle-ci inversée (ex : "camion" -> "noimac").

Solutions : verlan.py

Fonctions

- Possibles directement sans créer de classe.
- Peuvent retourner ou non un résultat.
- Peuvent être récursives.
- Exemple :

```
def add(a,b):
  return a+b
...
c=add(3,5)
print(c) # 8
```

Portée des variables (*scope*)

- Une variable définie en dehors d'une fonction a une portée globale.
- Une variable définie dans une fonction a une portée locale à cette fonction.
- Dans une fonction, les variables locales masquent automatiquement les variables globales de mêmes noms. Le mot-clé `global` permet d'accéder à une variable globale sans la masquer.
- Exemple :

```
g1='script-g1'
g2='script-g2'
def f():
  global g1
  g1='f-global-g1'
  g2='f-g2'
  g3='f-g3'
  print(g1,g2,g3)
f()          # f-global-g1 f-g2 f-g3
print(g1,g2) # f-global-g1 script-g2
#print(g3)   # Erreur
```

TD : fonctions récursives

- Transformer le TD Factorielle (`facto.py`) pour qu'il utilise une fonction récursive.

Solution : factoRec.py

- Python est-il limité en récursivité ?

Solution :

Fonctions

Arguments par défaut

- Exemple :

```
def supprimer(id,forcer=False):
  ...

supprimer(3)
supprimer(8,True)
```

Arguments nommés (*named arguments*)

- Exemple :

```
def add(colonne,rangee,dim=1):
  ...

add(rangee=3,colonne=5)
add(rangee=2,colonne=4,dim=2)
```

Fonctions

rest parameters

**args*

- Il reçoit tous les arguments positionnés **dans un tuple**. Exemple :

```
def supprimer(*ids,forcer=False):
  ...

supprimer(3,8,5,4)
supprimer(3,8,forcer=True)
```

***args*

- Il reçoit tous les arguments nommés **dans un dictionnaire**. Exemple :

```
def add(dim=1,**args):
  ...

add(rangee=2,colonne=4,dim=2)
```

Fonctions

Paramètres ou arguments ???

- Deux noms pour distinguer **OÙ** se trouvent les données.
- **Paramètre** (*parameter*, *formal parameter*, ***compile time*** *parameter*)
  = donnée attendue par la fonction donc connue à la compilation.
- **Argument** (*argument, actual parameter, **runtime** parameter*)
  = donnée envoyée lors de l'appel donc connue à l'exécution.

Passage par référence ou par valeur ???

- Le passage a toujours lieu **par référence**, mais les variables de types **immutables** ne sont pas modifiées donc tout se passe comme avec un passage par valeur.

Surcharge ???

- La surcharge de fonctions (ou méthodes) n'est pas possible puisque le typage n'est pas statique.

TD : passage de mutables et d'immutables

- Ecrire un programme qui démontre les différences entre le passage d'un argument mutable et d'un autre immutable.

Solution : passage.py

Modules

Importation d'un module (lui-même)

- Importe le module lui-même. Exemple :

```
import math # Seul "math" est défini.
...
print(math.sqrt(16)) # 4.0
print(math.pi)       # 3.141592653589793
```

Importation sélective d'éléments d'un module

```
from math import sqrt # "sqrt" est définie
                      # mais pas "math".
...
print(sqrt(16)) # 4.0
print(pi)       # Erreur !
print(math.pi)  # Erreur !
```

Modules

Quelques précisions...

- Tous les fichiers `.py` sont des modules, y compris le programme principal.
- Recherche des fichiers d'abord dans le *Module Search Path* puis dans le répertoire de l'appelant.
- Les modules importés sont conservés en cache après compilation.

 Un module Python NE correspond NI à un package Java NI à un module Ruby : c'est une simple ventilation du code en plusieurs fichiers.

 Il existe `from monModule import *` mais dangereux (sauf pour les modules de classes, voir plus loin).

TD

Carré magique

- Ecrire un programme qui demande une dimension (supérieure ou égale à 2) puis affiche une solution de carré magique dans la dimension saisie ou un message s'il n'y a pas de solution.
- RAPPEL
  Un carré magique de dimension N est un quadrillage dans lequel chaque rangée, chaque colonne et les 2 grandes diagonales, contiennent les nombres de 1 à N sans doublon.

Solution : carMag.py

POO

Rappel : objets, classes, instances

- Un objet contient des membres.
- Deux types de membres : propriétés et méthodes.
- Une propriété est une information.
- Une méthode est une capacité d'action.
- L'ensemble des propriétés constitue l'état de l'objet.
- Une classe est un moule pour créer des objets dits instances de la classe.
- Dans chaque classe, une méthode dite constructeur permet de créer des instances.
- Chaque instance d'une même classe possède ses propres valeurs de propriétés et partage les méthodes avec les autres instances.
- Il existe toutefois des propriétés de classe dont les valeurs sont communes à l'ensemble des instances. Elles sont dites statiques (*static*).
- De même, il existe des méthodes de classe qui n'agissent pas sur les instances mais sur la classe elle-même. Elles sont dites statiques (*static*).

POO

Classes

- Exemple :

```
class Vehicule:
  pass # Le mot-clé "pass" ne fait rien...

v=Vehicule() # Crée une instance de Vehicule.
```

POO

Propriétés dynamiques

- Des propriétés peuvent être ajoutées dynamiquement aux instances d'une classe. Exemple :

```
v1=Vehicule()
v1.couleur="rouge"
v2=Vehicule()
v2.nbRoues=4
print(v1.couleur)# rouge
print(v1.nbRoues)# Erreur !
print(v2.couleur)# Erreur !
print(v2.nbRoues)# 4
```

Dans cet exemple, la classe `Vehicule` et ses instances `v1` et `v2` ne possèdent pas les mêmes propriétés. **Technique dangeureuse, à fuir !**

POO

Variables de classe

- Elles sont déclarées à la racine de la classe.
- Elles sont accessibles via le nom de la classe.
- Exemple :

```
class Vehicule:
  nb=2

print(Vehicule.nb) # 0 (variable de classe)
v=Vehicule()
v.nb=3             # Variable dyn. d'instance.
print(v.nb)        # 3
print(Vehicule.nb) # 0
```

POO

Constructeur

- Il se nomme obligatoirement `__init__()` et **reçoit IMPLICITEMENT l'instance en cours de création** (mot-clé `this` dans la plupart des langages) libre de nom en Python et **traditionnellement appelée `self`**.
- Exemple :

```
class Vehicule:
  nb=0
  def __init__(self):
    Vehicule.nb+=1

v1=Vehicule()
v2=Vehicule()
print(Vehicule.nb) # 2
```

En fait, `__new__()`, qui reçoit obligatoirement la classe en premier argument, est appelée avant `__init__()`.

POO

Variables d'instance (non dynamiques)

- Elles sont obligatoirement déclarées dans le constructeur.
- Elles peuvent être passées au constructeur.
- Elles sont accessibles via le nom de l'instance.
- Exemple :

```
class Vehicule:
  nb=0
  def __init__(self,couleur=''):
    self.couleur=couleur
    Vehicule.nb+=1

v1=Vehicule("rouge")
v2=Vehicule()
print(v1.couleur) # rouge
print(v2.couleur) # (aucune couleur)
```

POO

Méthodes de classe

- Elles sont précédées du modificateur (*decorator*) `@classmethod`.
- Elles sont accessibles via le nom de la classe.
- **En premier argument, elles reçoivent IMPLICITEMENT la classe** dont le nom est libre mais **traditionnellement `cls`**.

```
class Vehicule:
  nb=0
  def __init__(self):
    Vehicule.nb+=1
  @classmethod
  def denombrer(cls):
    print("Nb. de véhicules : "+str(Vehicule.nb))

v1=Vehicule()
v2=Vehicule()
Vehicule.denombrer() # Nb. de véhicules : 2
```

POO

Méthodes d'instance

- Elles sont accessibles via le nom d'une instance.
- **En premier argument, elles reçoivent IMPLICITEMENT l'instance en cours de création traditionnellement appelée `self`**.
- Exemple :

```
class Vehicule:
  def __init__(self,couleur=''):
    self.couleur=couleur
  def afficher(self):
    print("Couleur : "+self.couleur)

v=Vehicule("rouge")
v.afficher()                # Couleur : rouge
Vehicule("bleu").afficher() # Couleur : bleu
```

POO

Membres privés, protégés...

- Ils n'existent pas.

Une classe est un objet

- Exemple :

```
class Voiture:
  pass

Auto=Voiture # Alias

v1=Voiture()
v2=Auto()
```

TD : POO

Boulangerie 1

- A l'aide du programme principal, de la sortie et des indications suivantes, écrire le code de l'ensemble des classes.
    - Main : boulangerie1/boul1.py
    - Sortie : boulangerie1/sortie1.txt

Les boulangers

- Ils ont un prénom.
- Ils fabriquent des produits.

Les produits

- Ils ont un nom, un prix de revient et un prix de vente.

Solution : boulangerie1

TD : POO

Boulangerie 2

- A l'aide du programme principal, de la sortie et des indications suivantes, modifier le code de l'ensemble des classes.
    - Main : boulangerie2/boul2.py
    - Sortie : boulangerie2/sortie2.txt

Les boulangers

- Ils entretiennent une liste de leurs **fabrications**.
- Ils éditent un bilan des **fabrications**.

Solution : boulangerie2

TD : POO

Boulangerie 3

- A l'aide du programme principal, de la sortie et des indications suivantes, modifier le code de l'ensemble des classes.
    - Main : boulangerie3/boul3.py
    - Sortie : boulangerie3/sortie3.txt

Les vendeuses

- Elles vendent des produits.
- Elles entretiennent une liste de leurs **ventes**.

Les boulangers

- Ils embauchent des vendeuses.
- Ils entretiennent une liste de leurs vendeuses.
- Ils éditent un bilan des fabrications et des **ventes**.

Solution : boulangerie3

POO

Rappel : héritage

- Une classe dite enfant (fille) peut hériter d'une classe dite parente (mère).
- Une classe fille hérite des membres de sa mère.
- Une classe fille peut comporter de nouveaux membres.
- Une classe fille peut redéfinir (*override*) les méthodes de sa classe mère, c'est à dire remplacer une méthode parente par une autre de même signature (même nom, mêmes paramètres).
- L'héritage correspond à une relation **EST UN** entre fille et mère, ou encore une relation de spécialisation/généralisation.
- Certains langages (dont Python) permettent l'héritage multiple (plusieurs parents). D'autres langages préfère le mécanisme des interfaces (Java...).

POO

Rappel : polymorphisme

- Du fait de l'héritage, chaque instance possède un type déclaré (fixe) et un type réel qui peut différer au cours de l'exécution.
- Le compilateur connait les types déclarés mais pas les types réels qui ne seront connus qu'au moment de l'exécution (*late binding*).
- La méthode exécutée est toujours celle correspondant au type REEL.
- Exemple en Java (la méthode `ecrire()` de la classe `Stylo` est redéfinie dans les classes filles `StyloBille` et `StyloPlume`) :

```
Stylos[] stylos={new Stylo(),
                 new StyloBille(),
                 new StyloPlume()};
for(Stylo stylo:stylos){
  stylo.ecrire();// Méthode déterminée à l'exécution.
}
```

POO

Duck-typing et polymorphisme

- Dans les langages sans typage statique, le polymorphisme n'existe pas puisqu'il n'y a pas de type déclaré.
- Parfois assimilé à tord au polymorphisme, le *duck-typing* garantit simplement que si une méthode donnée existe dans une classe ou dans l'une de ses classes parentes, alors toute instance de cette classe pourra appeler cette méthode avec succès quel que soit son type. C'est la méthode de la classe donnée ou, à défaut, de son plus proche parent qui sera exécutée.

POO

Héritage simple (*inheritance*)

- La fonction *top-level* `super()` permet d'accèder explicitement aux membres des parents.
- Exemple :

```
class Media:
  def __init__(self,titre=''):
    self.titre=titre

class Livre(Media):
  def __init__(self,titre='',nbPages=''):
    super().__init__(titre)
    self.nbPages=nbPages

livre=Livre("Millenium",350)
print(livre.titre+" : "+str(livre.nbPages)+" p.")
# Millenium : 350 p.
```

POO

Appel implicite au constructeur parent

- En l'absence (et seulement en l'absence) de méthode `__init__()`, il y a appel implicite au constructeur parent avec transmission des arguments. Exemple :

```
class Mere:
  def __init__(self,prenom=''):
    print("Je suis la mère "+prenom)

class Fille(Mere):
  pass

Mere("Michèle")  # Je suis la mère Michèle
Fille("Noël")    # Je suis la mère Noël
```

POO

Redéfinition (*overriding*)

- Toute méthode peut être redéfinie dans une sous-classe.
  Exemple :

```
class Animal:
  def crier(self):
    print("???!")

class Chien(Animal):
  def crier(self):
    print("Ouah!")

Chien().crier() # Ouah!
```

POO

Héritage multiple

Attention aux ambiguïtés !

```
class Animal:
  def identifier(self):
    print("Je suis un animal")
class Volant:
  def identifier(self):
    print("Je vole")
class Oiseau(Animal,Volant):
  pass
class Coccinelle(Volant,Animal):
  pass
class Mouette(Oiseau,Volant): # =(Animal,Volant,Volant)
  pass                        # =(Animal,Volant)
class Pigeon(Volant,Oiseau):  # =(Volant,Animal,Volant)
  pass                        # Erreur, ambiguïté !
```

POO

Héritage multiple (suite)

```
Oiseau().identifier()     # Je suis un animal
Coccinelle().identifier() # Je vole
Mouette().identifier()    # Je suis un animal
```

POO

Importation et héritage

A la différence des modules Ruby, les modules Python ne sont pas déclarés (avec un mot-clé) : il s'agit simplement d'une ventilation du code en plusieurs fichiers. En conséquence, en Python, l'importation d'un module dans une classe parente n'emporte pas l'importation dans les classes filles.

POO

Interroger le type d'un objet

Fonctions *top-level*

type(*object*)	Retourne la classe de `object`.
isinstance(*object*,*classinfo*)	Retourne `True` si `object` est du type `classinfo` ou d'un de ses parents.

```
class A:
  pass
class B(A):
  pass

print(type(A()))         # <class '__main__.A'>
print(type(B()))         # <class '__main__.B'>
print(isinstance(A(),A)) # True
print(isinstance(A(),B)) # False
print(isinstance(B(),A)) # True
print(isinstance(B(),B)) # True
```

Egalité d'identité entre objets

Opérateur `is`

- Retourne True si les objets pointent vers la même adresse mémoire.
- `a is b` est équivalent à `id(a)==id(b)`.
- Exemple :

```
class A:
  pass

a1=A()
a2=a1
a3=A()
print(a1 is a2)       # True
print(a1 is a3)       # False
print(a2 is a3)       # False
print(type(A()) is A) # True
```

TD : POO

Boulangerie 4 (1/2)

- A l'aide du programme principal, de la sortie et des indications de la diapo suivante, modifier le code de l'ensemble des classes.
  - Main : boulangerie4/boul4.py
  - Sortie : boulangerie4/sortie4.txt

Solution : boulangerie4

TD : POO

Boulangerie 4 (2/2)

Les boulangeries

- Elles ont un nom.
- Elles embauchent des boulangers (même patissiers) et des vendeuses.
- Elles entretiennent une liste de leurs employés.
- Elles éditent un bilan des fabrications et des ventes.

Les patissiers

- Ils sont des boulangers spécialisés qui peuvent fabriquer des produits de boulangerie ou des patisseries.

Les patisseries

- Elles sont des produits spécialisés.
- Elles peuvent être ou non au beurre.

POO

Méthodes magiques

- Elles permettent d'implémenter des fonctionnalités particulières et de redéfinir des opérateurs ou des méthodes. Exemples :

`__lt__(self,other)`	Redéfinit l'opérateur `<`, utilisé par la fonction *top-level* `sorted()`.
`__eq__(self,other)`	Redéfinit l'opérateur `==`.
`__add__(self,other)`	Redéfinit l'opérateur `+`.
`__mul__(self,other)`	Redéfinit l'opérateur `*`.
`__and__(self,other)`	Redéfinit l'opérateur `and`.
`__or__(self,other)`	Redéfinit l'opérateur `or`.
`__int__(self)`	Redéfinit la méthode `int()`.
`__str__(self)`	Redéfinit la méthode `str()` utilisée par `print()` donc équivalente à `toString()` en Java.

TD : POO

Tri d'une liste complexe

Etape 1 / 2 : boulangerie 5

- Dans la classe `Vendeuse` du TD précédent :
  - Implémenter `__str()__()` pour que `print()` affiche le prénom.
  - Implémenter la méthode `__lt__()` qui classe les vendeuses selon la valorisation de leurs ventes (une vendeuse est "inférieure" à une autre si son chiffre d'affaire est inférieur à celui de l'autre vendeuse).

Solution : boulangerie5

TD : POO

Tri d'une liste complexe

Etape 2 / 2 : tester

- Vérifier le bon fonctionnement avec le programme suivant :

```
croissant=Produit("croissant",0.15,1.10)
painDeMie=Produit("pain de mie",0.35,2.50)
v1=Vendeuse("v1")
v2=Vendeuse("v2")
v3=Vendeuse("v3")
v1.vendre(croissant,50)
v1.vendre(painDeMie,10)  # (55+25=80)
v2.vendre(painDeMie,30)  # (75)
v3.vendre(croissant,100) # (110)
vendeuses=sorted([v1,v2,v3])
for vendeuse in vendeuses:
  print(vendeuse,end=", ") # v2, v1, v3
```

Exceptions

Capture

- Syntaxe complète (voir note ci-dessous) :

```
try:
  ... code à tenter ...
except [classeException as erreur,...]:
  ... traitement de l'exception ...
else:
  ... code si aucune exception ...
```

- Un et un seul `try`.
- Zéro, un ou plusieurs `except`.
- Zéro ou un `else`.
- `print(`*`erreur`*`)` affiche le message contenu dans `erreur`.

 Sans classe précisée, toutes les erreurs sont capturées !

Exceptions

Exemples d'exceptions prédéfinies (*builtins*)

`ZeroDivisionError`	Division par zéro.
`NameError`	Identifiant indéfini.
`IndexError`	Index indéfini dans une séquence.
`KeyError`	Clé indéfinie dans un dictionnaire.
`TypeError`	Type(s) incompatible(s).
`AttributeError`	Membre indéfini.

Exceptions

Capturer de la plus précise à la plus générale

- **MAUVAIS exemple :**

```
try:
  x=n/d
except Exception as err:  # Capturera TOUT
  print(err)
except ZeroDivisionError: # Ne capturera RIEN
  print("Division par zéro !")
else:
  print("Résultat : "+str(x))
```

Exceptions

raise

- Permet de déclencher une exception. Exemples :

```
try:
  ...
except:
  ...
  raise # Redéclenche une exception capturée.

raise NameError # Déclenche l'exception.
```

Exceptions

Classe d'exception personnalisée

- Possible de créer une classe d'exception personnalisée en étendant la classe `Exception`.

TD : exceptions

Boulangerie 6

- Dans la classe `Vendeuse` du TD précédent, créer une classe d'exception personnalisée :
  - Quand une vendeuse vend une quantité inférieure ou égale à zéro, l'exception doit se déclencher et afficher :

```
BoulangerieException: Quantité incorrecte
```

- Vérifier le bon fonctionnement avec le programme suivant :

```
p=Produit('p',1.0,2.0)
v=Vendeuse('v')
v.vendre(p,0)
```

Solution : boulangerie6