Apprenez_à_programmer_en_Python.md

Sommaire

Introduction à Python

Origines et caractéristiques

Origines

1. langage créé par l'hollandais Guido van Rossum en 1991 (première version) (**BDFL** : *Benevolent Dictator for Life*)
2. associé à une organisation à but non lucratif (*Python Software Foundation*) créée en 2001
3. nom choisi en hommage à *"Monty Python"*

Caractéristiques

1. langage de scripts **interprété, multiplateforme, fortement typé, et multi-paradigme** (*orientée-objet et fonctionnelle*)
2. facile à prendre en main et assez performant
3. bibliothèque standard très riche et beaucoup de bibliothèques et *frameworks* utilitaires :
   - des scripts systèmes riches et performants ;
   - analyse scientifique ;
   - *machine learning* et *deep learning* avec le framework *TensorFlow* ;
   - interfaces graphiques ;
   - applications réseaux ;
   - framework *back-end* tel que *Django* ;
   - ...
4. *tout est objet* en **Python** => modulaire et facilement extensible via des bibliothèques tierces
5. versions :
   - `2.x` : depuis 2001
   - `3.0.1` : depuis 2009 (casse la rétrocompatibilité)
   - `3.6.5` : version en 2018

Utilisation sous Linux

1. dans la console en mode interactive :
   - `python3`
   - `CTRL + D` pour terminer la session
2. interpréter des fichiers source **Python** (`.py`) avec l'interpréte **python3** : `python3 fichier.py`
3. `>>>` : le prompt de l'interpréteur
4. `...` : l'instruction en cours n'est pas terminée encore

Les variables

Types de base

1. **entiers** (*int*) :
   - base **décimale** : `1, -2, ...`
   - base **binaire** : `0b0000, 0b0000111000, ...`
   - base **hexadécimale** : `0x10, 0x2A, ...`
2. **nombres flottants** (*float*) :
   - notation **décimale** : `1.2, -25.46357, .5 (0.5), -.26 (-0.26)`
   - notation **scientifique** : `2e-5, -5E10`
3. **chaînes de caractères** (*string*) circonscrites par :
   - des **apostrophes** :
     a. *exemple* : `'ceci est un string'`
     b. utilisé pour ne pas échapper les `""` explicitement
   - des **guillemets** :
     a. *exemple* : `"ceci est un autre string"`

    b. utilisé pour ne pas échapper les `''` explicitement
  - des **triples guillemets** :
    a. *exemple* : `"""ceci est le pire bizarre"""`
    b. utilisé pour ne pas échapper les `""` et les `''` explicitement, mais aussi pour conserver les **espaces blancs** entre les caractères du string
  - des **triples apostrophes** :
    a. *exemple* : `'''ceci est un string chelou'''`
    b. *idem* que les `"""`
4. **booléens** (*booleans*) : `True` ou `False`

Variables

1. les **variables** sont **dynamiquement typées** en **Python**, au fur et à mesure de leurs affectations successives
2. **contraintes de nommage** :
   - **regex** : /^[a-zA-Z_][a-zA-Z0-9_]*$/
   - **sensible à la casse**.
3. **conventions de nommage** :
   - **classes** : *CamelCase*
   - **constantes** : *Majuscule* avec le séparateur _ (*snake_case*)
   - **toute autre variable** : *minuscule* avec le séparateur _ (*Snake_case*)
4. **syntaxe de définition d'une variable** : `var_nom = valeur`
5. **opérateur d'affectation** :
   - `=`
   - enchaînable sur plusieurs variables de suite : `ma_var1 = ma_var2 = 10`
6. **opérateur de découpage d'une instruction sur plusieurs lignes** :

```
instruction \
reste
```

7. **commentaires** : `# commentaire monoligne`
8. **bloc d'instructions** : séquence d'instructions ayant une **indentation** permettant à l'interpréteur d'identifier leur **portée**

Opérateurs

Opérateurs arithmétiques

Symbole	Signification
+	addition
-	soustraction
*	multiplication
/	division réelle
//	division entière

Symbole	Signification
`%`	modulo
`**`	puissance

Opérateurs arithmétiques raccourcis

Symbole	Signification
`+=`	addition raccourcie
`-=`	soustraction raccourcie
`*=`	multiplication raccourcie
`/=`	division réelle raccourcie
`//=`	division entière raccourcie
`%=`	modulo raccourci

Opérateurs de comparaison

Symbole	Signification
`<`	strictement inférieur
`<=`	inférieur ou égal
`>`	strictement supérieur
`>=`	supérieur ou égal
`==`	égal à (égalité logique)
`!=`	différent de (différence logique)
`is`	égalité physique (égalité de référence)

Opérateurs logiques

Symbole	Signification
`and`	et logique
`or`	ou logique
`not`	non logique

Les structures conditionnelles

La structure conditionnelle if, elif, else

```
if expression_booleene:
  #instruction1 | bloc1 d'instructions
elif cond:
  #instruction2 | bloc2 d'instructions
else:
```

```
    #instructionSinon | blocSinon d'instructions
```

Les expressions ternaires

syntaxe :

```
expression1 if expression_booleene else expression2
```

une **expression conditionnelle** qui retourne :

1. la valeur de `expression1` si `expression_booleene` est `True`
2. sinon retourne la valeur de `expression2`

Exemple

```
annee = input("Veuillez saisir une année pour voir si elle est bissextile ou pas : ")
annee = int(annee)

if (annee%400 == 0) or (annee%4 == 0 and annee%100 != 0):
    print("année bissextile")
else:
    print("année non bissextile")
```

Les boucles

La boucle `while`

```
while expression_booleene:
  #instruction | blocInstructions
```

La boucle `for`

```
for element in sequence:
  #instruction | blocInstructions
```

L'opérateur `in`

tester si un **élément** est dans une **collection** ou **instancier** une **variable** depuis une **collection** dans une boucle `for`

`continue et break`

1. `break` : **instruction de rupture** de l'**itération courante** d'une boucle et la **sortie** de cette dernière
2. `continue` : **instruction de rupture** de l'**itération courante** d'une boucle et passage à l'**itération suivante**

La modularité

Fonctions

Introduction

Dans le **paradigme fonctionnel**, les **fonctions** sont des **éléments du premier plan** ("*first-class citizens*"), elles sont donc traitées comme toute expression ayant une valeur. Ceci est mis en oeuvre en **Python** où une **fonction** peut être :

1. **affectée à une variable** qui la référencera
2. utilisée comme **paramètre** dans une **autre fonction**
3. retournée comme **résultat** par une **autre fonction**

Remarque

En **Python**, les **fonctions** sont des **objets invocables** (*callable objects*) sur lesquels on peut appliquer l'opérateur d'invocation `()`

Syntaxe de déclaration d'une fonction

```
def nom_fonction([param1[, ...]]):
  #Bloc d'instructions
  #[return expression]
```

Éléments d'une fonction

1. les **paramètres** peuvent avoir des **valeurs par défaut** à la **C/C++**
2. on peut documenter une fonction via la *docstring* : une **description** de la **fonction** circonscrite par des `"""` et placée **au tout début** du bloc d'instructions définissant la fonction
3. lors du **passage des arguments** on peut **expliciter le nom du paramètre** auquel on passe un argument (*i.e.* utiliser un autre critère que la **position** :)
   - si on a **plusieurs paramètres** et qu'on souhaite ne pas prendre la tête avec leur ordre
   - **pas besoin de respecter l'ordre de déclaration des paramètres** de la fonction lors du passage des arguments
4. **signature** : *nom d'une fonction* => **impossible de surcharger une fonction** qui sera **écrasée par la nouvelle définition de la fonction** portant son nom

Exemple

```
def fonc(a=1, b=2, c=3, d=4, e=5):
  print("a =", a, ", b =", b, ", c =", c, ", d =", d, ", e =", e)

fonc() #a=1, b=2, c=3, d=4, e=5
fonc(4) #a=4, b=2, c=3, d=4, e=5
fonc(b=8, d=5) #a=1, b=8, c=3, d=5, e=5
fonc(b=35, c=48, a=4, e=9) #a=4, b=35, c=48, d=4, e=9
```

Fonctions lambda

1. les **fonctions** créées avec l'instruction `def` doivent être **référencés par des identificateurs** et **peuvent être référencées par des variables** => une **fonction doit posséder un nom**
2. les **fonctions lambdas** permettent de créer des **fonctions anonymes** qui **ne possèdent pas de noms** et qui **peuvent être référencées par des variables** pour leur **exécution ultérieure** ou **passées en argument** d'une **autre fonction** => l'instruction `lambda` est une **expression renvoyant une référence vers la fonction lambda déclarée** et qui **pourra être affectée à une variable** (on parle d'*expressions fonctionnelles*)
3. **syntaxes** :

```
#syntaxe de création de fonctions lambdas
  lambda x[, ...]: #instruction | bloc d'instructions
```

```
#syntaxe de référencement d'une fonction lambda par une variable
  nom_var = lambda x[, ...]: #instruction | bloc d'instructions

#syntaxe d'invocation d'une fonction lambda référencée par une variable
  nom_var([arg[, ...]])
```

Exemple

```
square = lambda x: x*x
sum = lambda x, y: x+y

print(square(5))
print(square(-18))
print(sum(1, 2))
print(sum(-2, -48))
print(sum(1, -1))
print(sum(square(3), square(4)))
```

Fonctions variadiques

1. **définition** : fonction à nombre de paramètres **inconnu**
2. **syntaxes** :

```
def fonction(*parametres):
  #fonction variadique dont le nombre de paramètres positionnels est inconnu

def fonction(**parametres_nommes):
  #fonction variadique dont le nombre de paramètres nommés est inconnu

def fonction(*parametres, **parametres_nommes):
  #fonction variadique dont le nombre de paramètres positionnels et
  #non nommés est inconnu
```

remarques :

1. si l'on veut avoir des **paramètres positionnels obligatoires**, il faut qu'ils précédent les **paramètres** (*positionnels et non nommés*) dont le nombre est inconnu :
2. si l'on veut utiliser des **paramètres nommés obligatoires** dont le nombre est connu dans une fonction variadique, il faut que la liste de ces paramètres suit la liste des **paramètres positionnelss** dont le nombre est inconnu :

```
def fonction(param_po1, param_po2, *parametres_positionnels, **parametres_nommes):
  #instructions

def fonction(*parametres_positionnels, param_no1, param_no2):
  #instructions
```

Exemples

```
def multiplication_table(n, max=10):
    """Fonction affichant la table de multiplication par nb de 1*nb à max*nb
    (max=10 par défaut) max doit être >= 0"""

    i=0
    while i<max:
        print(n, "*", i + 1, "=", (i+1)*n)
        i += 1
    print("==============================")
```

```
i = 1
while i < 5:
    multiplication_table(i)
    i += 1
#voir la documentation docstring de la fonction multiplication_table
help(multiplication_table)
####################################

def fonction_inconnue(*parametres):
    """Test d'une fonction variadique"""

    print("J'ai reçu : {}.".format(parametres))

fonction_inconnue() #J'ai reçu ()
fonction_inconnue(33) #J'ai reçu (33,)
fonction_inconnue('a', 'e', 'f') #J'ai reçu ('a', 'e', 'f')
x = 3.5
fonction_inconnue(x, [4], '...') #J'ai reçu (3.5, [4], '...')
####################################

def fonction_inconnue(**parametres_nommes):
    """Fonction permettant de récupérer les paramètres nommés
    dans un dictionnaire"""
    print("J'ai reçu en paramètres nommés : {}".format(parametres_nommes))

fonction_inconnue() #J'ai reçu ()
fonction_inconnue(p=4, j=8) #J'ai reçu en paramètres nommés : {'p': 4, 'j': 8}
fonction_inconnue(1) #Exception TypeError levée car la fonction ne prend aucun paramètre non nommé
####################################

def afficher(*parametres, sep=' ', fin='\n'):
    """Fonction chargée de reproduire le comportement de la fonction print()

    Elle doit finir par faire appel à print pour afficher le résultat, mais
    les paramètres devront déjà été formatés.
    On doit passer à print() une unique chaîne,
    en lui spécifiant de ne rien mettre à la fin"""

    parametres = list(parametres)
    for i, parametre in enumerate(parametres):
        parametres[i] = str(parametre)

    chaine = sep.join(parametres)
    chaine += fin
    print(chaine, end='')

afficher("dude", "leave me alone", 12, 14) #dude leave me alone 12 14
####################################

list = [1, 2, 3, 4]
print(*list) #1 2 3 4

parametres = {"sep": "  << ", "end": "-\n"}
print("Voici", "un", "exemple", "d'appel", **parametres) #Voici  << un  << exemple  << d'appel-
```

Modules

Éléments d'un module

1. **définition** : un **module** est un **ensemble** de **fonctions**, de **variables** et de **classes** (appelés *éléments*) d'un **même fichier** qui peuvent (*parfois doivent*) posséder une **relation particulière entre eux**
2. **utilisation** : pour utiliser les éléments d'un module, il faut **importer** ce dernier dans notre programme
3. remarque : certains modules de **Python** sont **importés automatiquement** lors de l'interprétation d'un script, d'autres ne le sont pas

4. **syntaxes** :

```
#syntaxe d'importation
import nom_module

#syntaxe d'utilisation d'un élément d'un module importé
nom_module.nom_var | nom_module.nom_fonction([args])
```

Exemple d'un module : le module `math`

1. non importé automatiquement par l'interprète : `import math`
2. description : un module de **fonctions mathématiques**

Constantes

- `math.e` : la **constante** `e = 2.7182818...`
- `math.pi` : la **constante** `pi = 3.1415...`

Fonctions

- `fabs(n) -> float` : renvoie la **valeur absolue** du **nombre** `n`
- `pow(base, exponent) -> float` : renvoie la valeur de `base^exponent` (renvoie toujours un **flottant**, inversement à `**` qui renvoie un **entier** quand il peut)
- `sqrt(n) -> float` : renvoie la **racine carré** du **nombre** `n`
- `exp(n) -> float` : renvoie `e^n`
- `ceil(n) -> int` : renvoie la **valeur plafond** de `n`
- `floor(n) -> int` : renvoie la **valeur plancher** de `n`
- `trunc(n) -> int` : renvoie la **valeur entière** de `n`
- `sin(theta)` : renvoie le **sinus** de l'**angle** `theta` (*en radians*)
- `asin(n) -> rad` : renvoie le **arcsinus** de `n` (*en radians*)
- `cos(theta)` : renvoie le **cosinus** de l'**angle** de `n` (*en radians*)
- `acos(n) -> rad` : renvoie le **arccosinus** de `n` (*en radians*)
- `tan(theta)` : renvoie la **tangente** de l'**angle** `theta` (*en radians*)
- `atan(n) -> rad` : renvoie le **arctangente** de `n` (*en radians*)
- `radians(theta) -> rad` : renvoie la **valeur en radians** de l'**angle** `theta` (*en degré*) (`1 rad = 57.29 degrés`)
- `degrees(theta) -> deg` : renvoie la **valeur en degrés** de l'**angle** `theta` (*en radians*)

Espaces de noms

1. chaque **importation** d'un **module** crée un **espace de noms propre** possédant *par défaut* le **même nom** que le **module importé**
2. chaque **élément** du **module importé** est dans la **portée définie par l'espace de noms correspondant**
3. **utilité** : on peut **créer des éléments** ayant les **mêmes noms** que ceux d'un **module importé** dans l'**espace de noms global** associé à notre **script**, sans risque d'avoir des **conflits de noms**

Différents types d'importation

```
#importation d'un module
#dont le nom de l'espace de noms créé et y correspondant est le même
import nom_module

#importation d'un module
```

```
#dont le nom de l'espace de noms créé et y correspondant est différent
import nom_module as nom_espace_noms_personnalisé

#importation d'un élément d'un module
#et l'associer à l'espace de noms courant
from nom_module import nom_fonction | nom_variable

#importation de tous les éléments d'un module
#et les associers à l'espace de noms courant
from nom_module import *
```

Définition de modules

Processus général de définition

1. commencer par indiquer **au début de fichier** le **chemin de l'interpréte Python** utilisé pour **interpréter le script** désignant le **module** :
   - `#!/usr/bin/python3` , appelé la *shebang* du **script**
   - le **chemin de l'interpréte** choisi sera pris par le **chargeur** de l'**OS** créant le **processus** associé à l'**exécution du script**
2. **documentation d'un module** :
   - on peut **documenter un module** par une *docstring* définie **au tout début du fichier désignant le module** (*après la shebang*)
   - on peut **documenter chaque fonction du module** en définissant sa *docstring* associée au tout début de son bloc d'instructions
3. **définir** les **variables et fonctions du module**
4. **tester le module** :
   - *soit* en l'**important ou une partie de ces éléments** dans d'autres scripts souhaitant les utiliser
   - *soit* en le testant en **lui-même** en utilisant la **variable prédéfinie** `__name__` :
     a. **variable créée automatiquement par l'interpréte** et contenant le **nom du thread courant**
     b. à chaque fois que le **module** sera **importé**, **tout son contenu est importé** et les **éventuelles instructions y contenues sont exécutées**. Pour conditionner l'exécution de ces instructions de test, on utilisera cette variable.
     c. si `__name__ = "__main__"` dans le **module** donc le **thread courant** est le **thread principal**. *Autrement dit*, le script désignant le **module** est **exécuté** depuis le **thread principal** en le lançant via la **console**. Et c'est ainsi qu'on souhaite exécuter les instructions de test
     d. sinon, le **script désignant le module n'est pas exécuté directement depuis la console**, mais utilisé pour **importer le module désigné**. Dans ce cas, le script désignant le **module n'est pas exécuté par le thread principal** et on ne souhaite pas ainsi exécuter les instructions de test relatives au **module**. En effet, on ne souhaite que **récupérer les éléments du module** et les **utiliser ailleurs**.

Le fichier `__pycache___` (Depuis `Python 3.2`)

Lors de l'**importation** d'un **module personnalisé**, **l'interprète Python** crée/cherche un répertoire `__pycache___` contenant un **code intermédiaire désignant le module** et permettant d'**accélérer le processus d'interprétation du code du module**

Exemples

```
#importation du module math
#sans changer le nom de l'espace de noms correspondant créé
import math
print(math.sqrt(16))
#########################################################

#importation du module math
#en changeant le nom de l'espace de noms correspondant créé
import math as Math
```

```
print(Math.sqrt(16))
##########################################################

#importation de la fonction retournant
#la valeur absolue d'un nombre depuis le module math
from math import fabs
print(fabs(-15))
print(fabs(0))
print(fabs(15))
##########################################################

#définition d'un module multiply.py contenant la fonction table()
#!/usr/bin/python3

"""module de multiplication contenant la fonction table"""

def table(n, max=10):
    """fonction affichant la table de multiplication de n de 1*n à max*n
    max = 10 (par défaut)
    """
    i = 1
    while i <= max:
        print(n, "*", i, "=", n*i)
        i += 1
    print("======================")

if(__name__ == "__main__"):
    table(4)
##########################################################

#importation et utilisation de la fonction table() dans un autre script
#!/usr/bin/python3
from multiply import *

table(3, 20)
##########################################################
```

Packages

Théorie

1. **définition** : un **package** sert à **regrouper thématiquement des modules et d'autres sous-packages**
2. **utilité** : **organiser** et **hiérarchiser** les **modules** et **minimiser les risques de conflits de nom d'éléments**

Pratique

1. les **packages** sont des **répertoires** pouvant **contenir d'autres sous-packages** (*répertoires*) ou des **modules** (*fichiers*)
2. **règles de création de packages** :
   - créer un **répertoire** désignant le **package** pouvant contenir :
     a. des **modules** : **fichiers à extension** `.py`
     b. des **sous-packages** : des **dossiers** pouvant contenir d'**autres sous-packages** ou **modules**
   - **contraintes de nommage** : *idem* que celles des **variables** en **Python**
3. exemples d'importation de packages :

```
#importer un package
import nom_package

#importation d'un module d'un package
from nom_package import nom_module
```

```
#importation d'un élément d'un module d'un sous-package d'un package
from nom_package.nom_sous_package.nom_module import nom_element
```

Exemples

```
#hiérarchie des packages :
# - test_package.py : module souhaitant utiliser la fonction "table()" du module "fonctions" du package "pa
# - package : package
  # - fonctions.py #module contenant la fonction "table()"

#importation de la fonction table() dans le fichier test_package.py
#de deux manières
from package import fonctions #importation du module
fonctions.table(5)
############################################################

from package.fonctions import table #importation de la fonction
table(5)
############################################################
```

Les exceptions et les assertions

Les exceptions

Introduction

1. **définition** : une **exception** est un **mécanisme** permettant de **lever des erreurs** et **éventuellement de les capturer et de les traiter** pour **éviter les arrêts brusques d'un programme et associer des traitements spécifiques au cas d'erreurs souhaités**
2. **en Python** : les **exceptions** sont **désignées** par :
   - un **type** désignant le **type de l'erreur levée** ;
   - un **message d'erreur** décrivant **l'erreur levée**
3. **exemples d'exceptions** :
   - `NameError` : erreur survenue lors de **l'utilisation d'un élément de programmation non défini** (*variables, fonctions, classes, ...*)
   - `ZeroDivisionError` : erreur survenue lors de la **division par zéro**
   - `TypeError` : erreur survenue lors de **l'utilisation d'un opérateur non défini pour un type**
   - `ValueError` : erreur survenue lors d'une **conversion de types**
   - `AssertionError` : erreur survenue lors du **non respect d'une assertion**

Le bloc `try-except-else-finally`

1. **définition** : des **blocs** utilisés pour **capturer une erreur et la traiter** en exécutant des actions spécifiques
2. **syntaxe** :

```
try:
  # bloc à essayer

#il est préférable de spécifier le type de l'exception à capturer,
#sinon python capturera n'importe quel erreur et la traitera dans ce bloc
except [typeException1[ as exceptionMessage1]]:
  #bloc qui sera exécuté en cas d'exception de type typeException1 | pass
  #pass : mot-clé utilisé indiquant de ne rien faire en cas de capture d'une erreur)
  #on peut utiliser exceptionMessage1 pour afficher le message généré
```

```
    #lorsque l'exception de type typeException1 est levée

  [except [typeException2[ as exceptionMessage2]]]:
    #bloc qui sera exécuté en cas d'exception de type typeException2  | pass

  [...]

  [except [typeExceptionN[ as exceptionMessageN]]]:
    #bloc qui sera exécuté en cas d'exception de type typeExceptionN  | pass]

  [else:
    #bloc qui sera exécuté si aucune exception n'est levée
    #généralement ce bloc peut être mis dans le bloc try
    #mais on préfère l'utiliser à part pour séparer
    #les instructions de la clause try
    #pouvant générer une erreur et celles les suivant ne pouvant pas en générer]
  ]

  [finally:
    #bloc qui sera exécuté qu'il y aura des exceptions levées ou non
  ]
```

Lever une exception

1. **définition** : **mécanisme** consistant à **générer une erreur logicielle dans un contexte bien défini**
2. **syntaxe** :

```
raise typeException("message à afficher")
```

Exemple

```
numerateur = 10
denominateur = 5

try:
    resultat = numerateur/denominateur
except NameError as message: #levée si numerateur ou denominateur non défini
    print("erreur : ", message)
except TypeError as message: #levée si numerateur ou denominateur posséde un type incompatible avec l'opéra
    # print("erreur : ", message)
    pass #capturer cette exception sans rien faire
except ZeroDivisionError as message: #levée si l'on divise par 0 (si denominateur = 0)
    print("erreur : ", message)
else:
    print("le résultat obtenu est ", resultat)
finally:
    print("I will appear no matter what...")
```

Les exceptions et l'héritage

(*cf. Chapitre 14 - L'héritage*)

1. les **exceptions** sont des **classes hiérarchisées** dans une **relation d'héritage**, héritant toutes de la **classe** `BaseException`
2. dans une **clause** `except typeException1` : on peut **capturer l'exception** de type `typeException1` et n'importe quelle **exception** de type **héritant** de celui de `typeException1`
3. consulter l'**hiérarchie d'héritage d'une classe d'exception** : `help(typeException)` (*uniquement les superclasses built-in*)

Créer ses propres exceptions

(*cf. Chapitre 14 - L'héritage*)

1. nos **exceptions doivent hériter** de l'une des deux **classes d'exceptions** proposées par **Python** (*ou d'autres plus spécialisées, selon le contexte*) :
   - `BaseException` :
     a. la **superclasse** de toutes les **exceptions** en **Python**
     b. les **exceptions** ne sont pas forcément des **cas d'erreur**, mes des **cas d'interruptions bien spécifiques** (*e.g.* `KeyboardInterrupt` qui est levée quand on **tappe** `CTRL + C`)
   - `Exception` :
     a. la **superclasse** de toutes les **exceptions d'erreurs** en **Python** qui **hérite** de la **classe** `BaseException`
     b. les **exceptions** sont forcément des **cas d'erreur**
     c. la **classe** la plus souvent **héritée** pour **créer des exceptions personnalisées**
2. nos **exceptions** doivent **contenir** :
   - un **constructeur** : **initialisant le message d'erreur**, et **éventuellement d'autres attributs**
   - une **implémentation** de la **méthode spéciale** `__str__()` pour **afficher l'erreur**, son **message** et **éventuellement d'autres informations** lorsqu'elle est levée

Exemple

```
class MonException(Exception):

    #constructor
    def __init__(self, fichier, ligne, message):
        self.fichier = fichier
        self.ligne = ligne
        self.message = message

    #methods
    def __str__(self):
        return "[{}:{}]: {}".format(self.fichier, self.ligne, self.message)

raise MonException("plop.conf", 34, \
"Il manque une parenthèse fermante à la fin de l'expression")
```

Les assertions

1. **définition** : un **mécanisme** permettant de **vérifier** des **prédicats** sur :
   - les **invariants de classes, de fonctions, de flux, ...**
   - les **axiomes des types abstraits de données**
   - les **post-conditions**
2. **fonctionnement** :
   - utilisé en général dans des **blocs** `try-except-else-finally`
   - si le **prédicat** est **vérifié** renvoie `True`
   - sinon, une **exception** `AssertionError` est **levée**
3. **syntaxe** : `assert expression_booleene`

Exemple

```
#!/usr/bin/python3
def estBissextile(annee):
    return annee%400 == 0 or (annee%4 == 0 and annee%100 !=0)

annee = input("Veuillez saisir une année pour savoir" \
```

```
+ " si elle est bissextile ou pas : ")
try:
    annee = int(annee)
    assert annee>0
except ValueError as message :
    print("Erreur : ", message)

if estBissextile(annee):
    print(annee, ": bissextile")
else:
    print(annee, ": non bissextile")
```

TP : Tous au ZCasino

```
#!/usr/bin/python3
from random import randrange
from math import ceil

def init_somme_argent(n):
    print("Vous rentrez dans le casino avec ", n, "euros")
    return n

def saisir_nombre_pari(msg, min, max):
    pari_user = -1
    while pari_user <0:
        try:
            pari_user = input(msg)
            pari_user = int(pari_user)
            assert pari_user >= min and pari_user < max
        except ValueError as message:
            print("Erreur : ", message)
            pari_user = -1
            continue
        except AssertionError:
            print("Erreur : le nombre", pari_user, \
            "n'est pas dans l'intervalle [", min, ", ", max, "]")
            pari_user = -1
            continue
    return pari_user

def saisir_somme_pari(msg, somme):
    somme_user = 0
    while somme_user == 0:
        try:
            somme_user = input(msg)
            somme_user = int(somme_user)
            assert somme_user > 0 and somme_user <=somme
        except ValueError as message:
            print("Erreur : ", message)
            somme_user = 0
            continue
        except AssertionError:
            print("Erreur : la somme mise", somme_user, \
            "n'est pas entre 0 et ", somme)
            somme_user = 0
            continue
    return somme_user

def get_couleur(n):
    if n%2 == 0:
        return "noir"
    else:
        return "rouge"
```

```
MIN = 0
MAX = 50
continuer = True

somme = init_somme_argent(1000)
while continuer == True and somme > 0 :
    nombre_user = saisir_nombre_pari("Veuillez saisir un numéro entre " \
    + str(MIN) + " et " + str(MAX-1) + \
    " sur lequel vous mettrez une somme : ", MIN, MAX)
    somme_user = saisir_somme_pari("Veuillez saisir la somme " \
    + "que vous souhaiter mettre (>0) : ", somme)

    nombre = randrange(MIN, MAX)
    if nombre_user == nombre:
        print("Vous avez gagné !")
        print("Vous avez parié sur ", nombre_user, "(", \
        get_couleur(nombre_user), ") et le nombre obtenu est ", \
        nombre, "(", get_couleur(nombre), ")")
        somme += somme_user *3
        print("Vous avez maintenant ", somme, "euros !")
    elif get_couleur(nombre_user) == get_couleur(nombre):
        print("Vous avez gagné !")
        print("Vous avez parié sur ", nombre_user, "(", \
        get_couleur(nombre_user), ") mais le nombre obtenu est ", \
        nombre, "(", get_couleur(nombre), ")")
        somme += ceil(somme_user / 2)
        print("Mais votre bille pariée a la même couleur que la bille tirée." \
        + "Vous avez maintenant ", somme, "euros !")
    else:
        print("Vous avez perdu !")
        print("Vous avez parié sur ", nombre_user, "(", \
        get_couleur(nombre_user), ") et le nombre obtenu est ", \
        nombre, "(", get_couleur(nombre), ")")
        somme -= somme_user
        print("Vous avez maintenant ", somme, "euros !")

    if somme == 0:
        continuer = False
    else:
        reponse = input("Voulez vous continuer la partie (O/N) ?")
        continuer = True if reponse == 'O' else False

    if continuer == False:
        print("Au revoir !")
```

Les structures de données principales

Les chaînes de caractères

La classe `str`

1. **définition** : un **string** est une **séquence de caractères immuable**
2. `len(str)` : fonction renvoyant la **longueur** d'une **séquence**, en particulier d'un **string**

Méthodes

- `str() -> str` : **constructeur par défaut** renvoyant un **string vide** (*équivalent à* `""`)

- `str("valeur") -> str` : **constructeur paramétré** renvoyant un **string initialisé à** `valeur` (*équivalent à* `"valeur"`)
- `S.lower() -> str` : renvoie une **copie** de `S` **en minuscule**
- `S.upper() -> str` : renvoie une **copie** de `S` **en majuscule**
- `S.capitalize() -> str` : renvoie une **copie** de `S` ayant la **première lettre en Majuscule** et le **reste de ces lettres en minuscule**
- `S.strip() -> str` : renvoie une **copie** de `S` n'ayant **aucun caractère blanc en début ou en fin**
- `S.isdigit() -> bool` : renvoie `True` si `S` correspond à un nombre, sinon renvoie `False`
- `S.isalpha() -> bool` : renvoie `True` si **toutes les lettres** de `S` sont **alphabétiques**, sinon renvoie `False`
- `S.isalnum() -> bool` : renvoie `True` si **toutes les lettres** de `S` sont **alphanumériques**, sinon renvoie `False`
- `S.center(length) -> str` : renvoie une **copie** de `S` de **longueur** `length` et dont le **contenu** est **centré** et **entouré** par des **espaces blancs**
- `S.format(x[=v0], y[=v1], ..., z[=vN]) -> str` :
  i. `S = <str0> + "{[index]}" + <str1> + "{[index]}" ... + {[index]}`
  ii. `S = <str0> + "{[x]}" + <str1> + "{[y]}" ... + {[z]}`
  iii. renvoie une **copie** de `S` **formatée** selon les **paramètres** du **string de formatage** passé en **argument**
- `S.count(<substring>, start=0, end=len(S)) -> int` : renvoie le **nombre d'occurrences** de `substring` dans `S` [de `start` à `end`]
- `S.find(<substring>, start=0, end=len(S)) -> int` : renvoie **l'index de la première occurrence** de `substring` dans `S` [de `start` à `end`] sinon renvoie -1
- `S.replace(<substring>, new[, max]) -> str` : **remplacer** [*au plus* `max` **occurrences**] de `substring` dans `S` par `new`
- `S.join(<list>) -> str` : **joindre** les **éléments** de la **liste** `list` par le **délimiteur** `S`
- `S.split(sep=' ') -> list of str` : **séparer** les **mots** du **string** `S` selon le **séparateur** `sep` dans une **liste de strings**
- `S.encode() -> byte string` : permet d'**encoder** le **string** `S` dans une **chaîne de bytes**

Opérateurs

- `+` : **opérateur de concaténation** de **strings**
- `+=` : **opérateur raccourci de concaténation** de **strings**
- `[i]` : **opérateur d'accès à un caractère d'un string** (*en lecture uniquement*) :
  i. *si* `0 <= i < len(s)` , alors **renvoyer** `str[i]`
  ii. *sinon si* `-len(S) < i < 0` , alors **renvoyer** `str[len(s) + i]`
  iii. *sinon* l'**erreur** `IndexError` sera levée
- **opérateur d'accès au facteurs d'un string** :
  i. `str[min:max]` : renvoyer le **string** entre `min` (*inclus*) et `max` (*non inclus*)
  ii. `str[:max]` : renvoyer le **string** entre 0 et `max`
  iii. `str[min:]` : renvoyer le **string** entre `min` et `len(str)`

Exemples

```
chaine = "        dude leave me alone        "
chaine_maj = chaine.upper() #"        DUDE LEAVE ME ALONE        "
chaine_min = chaine.lower() #"        dude leave me alone        "
chaine_cap = chaine.capitalize() #"        dude leave me alone        "
chaine_sans_espaces = chaine.strip() #"dude leave me alone"
chaine_centered = chaine.strip().center(20) #"     dude leave me alone      "
##########################################

prenom = "Paul"
```

```
nom = "Dupont"
age = 21
print( \
  "Je m'appelle {0} {1} ({3} {0} pour l'administration) et j'ai {2} "\
  "ans.".format(prenom, nom, age, nom.upper())
)
################################################

date = "Dimanche 8 juillet 2018"
heure = "18:38"
print("Cela s'est produit le {}, à {}.".format(date, heure))
################################################

adresse = """
{no_rue}, {nom_rue}
{code_postal} {nom_ville} ({pays})"""
.format(no_rue=6, nom_rue="rue des Postes", \
code_postal=75003, nom_ville="Paris", pays="France")
print(adresse)
################################################

prenom = "Paul"
message = "Bonjour"
age = 21
chaine_complete = message + " " + prenom + " qui a " + str(age) + " ans."
print(chaine_complete)
################################################

str = "when a dude meets a dudette, " \
+ "the chances are they will bring small dudes and dudettes " \
+ "if they are drunk"
print(str)
print("le nombre d'occurrences de 'du' dans le string est ", str.count("du"))
print("l'indexe de la première occurrence de 'dude' dans le string est", \
 str.find("dude"))
print("en remplaçant seulement les deux premières occurrences de " \
+ "'dude' par 'rude' " \
+ "dans le string on obtient le string :", \
str.replace("dude", "rude", 2))
################################################

def afficher_flottant(flottant):
    """Fonction prenant en paramètre un flottant et
    renvoyant une chaîne de caractères représentant la troncature de ce nombre.
    La partie flottante doit avoir une longueur maximum de 3 caractères.
    De plus le point décimal sera remplacé par une virgule"""

    if type(flottant) is not float :
        raise TypeError("Le paramètre attendu doit être un flottant")

    flottant = str(flottant)
    partie_entiere, partie_decimale = flottant.split(".")
    return ",".join([partie_entiere, partie_decimale[:3]])

print(afficher_flottant(3.9999998))
print(afficher_flottant(1.5))
```

Les listes, tuples et intervalles

La classe `list`

1. **définition** : en **Python**, les **listes** sont des **séquences désignant des tableaux hétérogènes dynamiques ordonnés et mutables**

2. `len(l)` : fonction renvoyant la **longueur d'une séquence**, en particulier d'une **liste**
3. `enumerate(l)` : fonction renvoyant un **ensemble de tuples** contenant **chacun l'élément courant** de la **liste** `l` et son **indice** (*indice suivi de l'élément*)
4. `sorted(l, reverse=False[, ...])` : fonction renvoyant une **copie** de la **liste** `l` mais qui est **triée par ordre croissant** (*par défaut*) selon l'**ordre naturel** défini sur les **éléments** de `l`
5. **compréhension de listes** :

```
nouvelle_liste = [instruction for element in liste [if expression_booleen]]
```

Méthodes

- `list() -> list` : **constructeur par défaut** renvoyant une **liste vide** (*équivalent à* `[]`)
- `list(v1, v2, ..., vN) -> list` : **constructeur paramétré** renvoyant une **liste contenant les éléments** `v1, v2, ..., vN` (*équivalent à* `[v1, v2, ..., vN]`)
- `l.append(element) -> None` : **ajoute** `element` **à la fin** de la **liste** `l`
- `l.insert(index, element) -> None` :
  i. si `0<= index < len(l)` alors **ajoute** `element` **à l'indexe** `index` de la **liste** `l`
  ii. si `-len(l) < index < 0` alors **ajoute** `element` **à l'indexe** `len(l) + index` de la **liste** `l`
- `l1.extend(l2) -> None` : **concaténe** la **liste** `l2` **en fin** de la **liste** `l1`
- `l.remove(element) -> None` : **supprime** la **première occurrence** de `element` dans la **liste** `l` (*s'il y appartient*), sinon ne fait rien
- `l.clear() -> None` : **supprime tous les éléments** de la **liste** `l`
- `l.count(element) -> integer` : renvoie le **nombre d'occurrences** de la **valeur** `value` dans la **liste** `l`
- `l.index(element) -> integer` : renvoie l'**indexe de la première occurrence** de `element` dans la **liste** `l`
- `l.pop([index]) -> element` : **supprime** l'**élément au dernier indexe** (*par défaut*) [*ou à l'index* `index`] et le **renvoie**
- `l.reverse() -> None` : **inverse** la **liste** `l`
- `l.sort(reverse=False[, ...]) -> None` : **tri** la **liste** `l` **par ordre croissant** selon l'**ordre naturel** sur ses **éléments**

Opérateurs

- `+` : **opérateur de concaténation de listes**
- `+=` : **opérateur raccourci de concaténation de listes**
- `[i]` : **opérateur d'accès à un élément d'une liste** (*en lecture et écriture*) :
  i. *si* `0 <= i < len(l)`, alors **renvoyer** `l[i]`
  ii. *sinon si* `-len(l) < i < 0`, alors **renvoyer** `str[len(l) + i]`
  iii. *sinon* l'**erreur** `IndexError` sera levée
- `del l[i]` :
  i. *si* `0 <= i < len(l)`, alors **supprimer** `l[i]`
  ii. *sinon si* `-len(l) < i < 0`, alors **supprimer** `str[len(l) + i]`
  iii. *sinon* l'**erreur** `IndexError` sera levée

Compréhension de listes (*list comprehensions*)

1. **définition** :
   - instruction ayant une syntaxe particulière permettant d'**appliquer un traitement à chaque élément d'une liste** et/ou de la **filtrer** selon une **condition**.
   - La **liste obtenue** est **renvoyée** par l'instruction (*i.e. la liste originale est immuable*)
2. **syntaxe** :

```
nouvelle_liste = [instruction for element in liste [if expression_booleene]]
#expression_booleene est facultative et est utilisé
#pour définir un prédicat permettant de filtrer la liste
```

La classe `tuple`

1. **définition** : en **Python**, les **tuples** sont des **séquences désignant des tableaux hétérogènes dynamiques ordonnées mais immuables**
2. **utilité** :
   - faire des **affectations multiples de variables**
   - **échanger des valeurs** entre des **variables**
   - **retourner plusieurs résultats** depuis une **fonction** sous forme d'un **tuple**
   - créer des fonctions attendant un **nombre inconnu** de **paramètres positionnels** qui seront traités comme un **tuple**
   - **transformer une liste de valeurs** en une **liste de paramètres d'une fonction**

```
#créer des fonctions attendant un nombre inconnu de paramètres positionnels
#qui seront traités comme un tuple
def fonction(*parametres):
  #code

#transformer une liste de valeurs en une liste de paramètres d'une fonction
#invocation de la fonction "fonction" avec les arguments obtenus
#en transformant la liste "parametres" en un tuple contenant ces arguments
fonction(*arguments)
```

Méthodes

- `() -> tuple` : **constructeur par défaut** renvoyant un **tuple vide**
- `(v1,[ v2[, ...[, vN]]]) -> tuple` : **constructeur paramétré** renvoyant un **tuple** contenant les **valeurs** (`v1[, v2[, ...[, vN]]]`)
- `var1[, var2[, ...[, varN]]] = val1[, val2[, ...[, valN]]]` :
  i. **affectations multiples** des **valeurs** `val1[, val2[, ...[, valN]]]` aux **variables** `var1[, var2[, ...[, varN]]]`
  ii. on peut **entourer** les **variables et les valeurs par des parenthèses**, mais ce n'est **pas nécessaire** (*l'interpète Python comprendra qu'il s'agit d'un tuple à partir des virgules séparant les identificateurs des variables*)

Les intervalles

En **Python**, un **intervalle** est une **séquence d'entiers immuable**, qui est souvent utilisée pour **définir un intervalle de valeurs** utilisable par une **variable d'indice** dans la **boucle** `for`

La classe `range`

1. **description** : une classe définissant un **intervalle** en **Python**
2. **constructeur** : `range(start=0, stop[, step=1])` : **créer et initialiser** un **intervalle d'entiers** commençant de `start` (*inclus*) à `stop` (*non inclus*) avec un **écart** de `step` entre les **valeurs** de l'**intervalle**

Exemples

```
#Exemple d'utilisation des méthodes de listes
```

```
liste = [1, 2, 3, 4]
print(liste) #[1, 2, 3, 4]

liste.append("dude")
print(liste) #[1, 2, 3, 4, "dude"]

liste.insert(2, "Hello there")
print(liste) #[1, 2, "Hello there", 3, 4, "dude"]

liste += [5, 6, 7, 8]
print(liste) #[1, 2, "Hello there", 3, 4, "dude", 5, 6, 7, 8]

liste.extend([9, 10])
print(liste) #[1, 2, "Hello there", 3, 4, "dude", 5, 6, 7, 8, 9, 10]

liste.clear()
print(liste) #[]

liste = [1, 2, 3]
print(liste) #[1, 2, 3]

del liste[0]
print(liste) #[2, 3]

liste.remove(2)
print(liste) #[3]

liste.extend([4, 5, 6, 7, 8, 3, 4, 5, 6, 7, 3, 4])
print(liste) #[3, 4, 5, 6, 7, 8, 3, 4, 5, 6, 7, 3, 4]
print("nombre de 3 dans la liste =", liste.count(3)) #3
print("l'indexe de la première occurrence de 3 dans la liste =", liste.index(3)) #0

element_a_index_3 = liste.pop(3)
print(liste) #[3, 4, 5, 7, 8, 3, 4, 5, 6, 7, 3, 4]
print("l'élément supprimé à l'indexe 3 est ", element_a_index_3) #6

dernier_element = liste.pop()
print(liste) #[3, 4, 5, 7, 8, 3, 4, 5, 6, 7, 3]
print("l'élément supprimé au dernier indexe est ", dernier_element) #4

liste.reverse()
print("la liste inversée : ", liste) #[3, 7, 6, 5, 4, 3, 8, 7, 5, 4, 3]

liste.sort()
print("la liste triée : ", liste) #[3, 3, 3, 4, 4, 5, 5, 6, 7, 7, 8]
################################################################

#Exemple d'application d'un traitement à chaque élément d'une liste
#en utilisant des compréhensions de listes
liste_origine = [0, 1, 2, 3, 4, 5]
liste_obtenue = [nb ** 2 for nb in liste_origine]
print(liste_obtenue) #[0, 1, 4, 9, 16, 25]
################################################################

#Exemple d'application de filtrage d'une liste
#sans application d'aucun traitement aux éléments filtrés
#en utilisant des compréhensions de listes
liste_origine = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
liste_obtenue = [nb for nb in liste_origine if nb%2 == 0]
print(liste_obtenue) #[2, 4, 6, 8, 10]
################################################################

#tri d'une liste de produits et de leurs quantités
#en utilisant la méthode sort() sur les quantités des produits
#et les compréhensions de listes
def tri_par_qte_decroissant(inventaire):
    inventaire_inverse = [(qte, produit) for (produit, qte) in inventaire]
```

```
    inventaire_inverse.sort(reverse=True)
    inventaire_triee = [(produit, qte) for (qte, produit) in inventaire_inverse]
    return inventaire_triee

inventaire = [
("pommes", 22),
("melons", 4),
("poires", 18),
("fraises", 76),
("prunes", 51)
]

print("l'inventaire trié est ", tri_par_qte_decroissant(inventaire))
```

Les dictionnaires

La classe `dict`

1. **définition** : en **Python**, les **dictionnaires** sont des **tableaux associatifs hétérogènes dynamiques non ordonnés** :
   - **associatif** : **ensemble de paires clé/valeurs**
   - **hétérogènes** : les **clés** peuvent avoir des **types différents** ainsi que les **valeurs**
   - **dynamiques** : les **paires clé/valeur** peuvent être **ajoutées et supprimées dynamiquement**
   - **non ordonné** : il n'y a pas la notion d'**indice** ni la notion d'**ordre** sur les **paires stockées**
2. les **clés** :
   - *peuvent* être **quasiment** de **n'importe quel type**
   - *doivent* être **uniques** dans le **dictionnaire**
3. les **valeurs** peuvent être de **n'importe quel type**

remarques :

```
#possibilité de récupérer les paramètres nommés
#d'une fonction variadique dans un dictionnaire
def fonction(**parametres) :
  #code

#possibilité de transformer un dictionnaire contenant des paires clés/valeurs
#en une liste de paramètres nommés d'une fonction.
#invocation de la fonction "fonction" avec les arguments obtenus
#en transformant le dictionnaire "parametres" en un tuple
#contenant ces arguments sous forme de "param=valeur"
fonction(**arguments)
```

Méthodes

- `dict() -> dict` : **constructeur par défaut** renvoyant un **dictionnaire vide** (*équivalent à* `{}`)
- `{k1: v1, k2: v2, ..., kN: vN} -> dict` : **constructeur paramétré** renvoyant un **dictionnaire contenant les paires clé/valeur** `k1 -> v1, k2 -> v2, ..., kN -> vN`
- `dict.pop(k[, valeur_erreur]) -> v` :
  i. *si* `k` est une **clé** dans le **dictionnaire** `dict` , alors **supprime** la **paire** `k/v` (`v` est la **valeur associée** à la **clé** `k`) et **renvoie** la **valeur** `v`
  ii. *sinon si* `valeur_erreur` est fournie, alors elle sera **renvoyée**
  iii. *sinon*, l'**erreur** `KeyError` sera levée
- `dict.clear() -> None` : **supprimer tous les éléments du dictionnaire** `dict`
- `dict.keys() -> set` : **renvoyer** un **ensemble** contenant les **clés** du **dictionnaire** `dict` **ordonné** par l'**ordre naturel**

sur les **clés** du **dictionnaire**

- `dict.values() -> list` : **renvoyer** une **liste** contenant les **valeurs** du **dictionnaire** `dict` **ordonnée** par l'**ordre de leurs ajouts au dictionnaire**
- `dict.items() -> list of tuples` : **renvoyer** une **liste** contenant les **paires clés/valeurs** du **dictionnaire** `dict` (*sous forme de tuples*) **ordonnée** par l'**ordre de leurs ajouts au dictionnaire**

Opérateurs

- `dict[key]` : **opérateur d'accès à une valeur d'un dictionnaire** via la **clé** `key` :
  i. **en lecture** : `dict[key]` :
    - *si* `key` est une **clé** dans le **dictionnaire** `dict` , la **valeur associée** sera **renvoyée**
    - *sinon* l'**exception** `KeyError` sera levée
  ii. **en écriture** : `dict[key] = valeur` :
    - *si* `key` est une **clé** dans le **dictionnaire** `dict` , la **valeur associée précédente** sera **écrasée** par la **nouvelle valeur** `valeur`
    - *sinon* la **paire** `key -> valeur` sera **rajoutée** au **dictionnaire**
- `del dict[key]` :
  i. *si* `key` est une **clé** dans le **dictionnaire** `dict` , la **paire** `key/value` (`value` est la **valeur associée** à la **clé** `key`) sera **supprimée** de `dict`
  ii. *sinon* l'**exception** `KeyError` sera levée

Méthodes de parcours

```
#parcours des clés
for key in dictionnary:
  #instruction | bloc d'instructions

for key in dictionnary.keys():
  #instruction | bloc d'instructions

#parcours des valeurs
for value in dictionnaire.values():
  #instruction | bloc d'instructions

#parcours des clés et valeurs simultanément
for key, value in dictionnaire.items():
  #instruction | bloc d'instructions
```

Exemples

```
#création d'un dictionnaire et d'ajout de valeurs
dictionnaire = {}
dictionnaire["pseudo"] = "dude"
dictionnaire["mot de passe"] = "aJkFsE12EsqehA4E5f5X"
print(dictionnaire) #{'pseudo': "dude", 'mot de passe': "aJkFsE12EsqehA4E5f5X"}

print("pseudo :", dictionnaire["pseudo"]) #pseudo : dude
dictionnaire["pseudo"] = "el duderino"
print("pseudo après :", dictionnaire["pseudo"]) #pseudo après : el duderino

print(dictionnaire) #{'pseudo': "el duderino", 'mot de passe': "aJkFsE12EsqehA4E5f5X"}
############################################################

#création d'un dictionnaire contenant les positions des pièces d'un échiquier
echiquier = {}
echiquier['a', 1] = "tour blanche" #la clé est un tuple ('a', 1) pointant vers la valeur "tour blanche"
```

```
echiquier['b', 1] = "cavalier blanc"
echiquier['c', 1] = "fou blanc"
echiquier['d', 1] = "reine blanche"
echiquier['a', 2] = "pion blanc"
echiquier['b', 2] = "pion blanc"
print(echiquier)
#{('a', 1): 'tour blanche', ('b', 1): 'cavalier blanc', ('c', 1): 'fou blanc',
#('d', 1): 'reine blanche', ('a', 2)! 'pion blanc', ('b', 2): 'pion blanc'}
############################################################

#différents parcours d'un dictionnaire
fruits = {"pommes": 21, "melons": 3, "poires": 31}
for key in fruits.keys():
    print("key :", key) #key: pommes\nkey : melons\nkey: poires

print("==========================") #==========================

for value in fruits.values():
    print("valeur :", value) #valeur: 21\nvaleur: 3\nvaleur: 31
print("==========================") #==========================

for key,value in fruits.items():
    print("{k} -> {v}".format(k=key, v=value))
    #pommes -> 21\nmelons -> 3\npoires -> 31
print("==========================") #==========================
```

Les fichiers

Schéma de manipulation des E/S

1. **ouvrir un fichier** avec `open(path, mode='r')` :

- `mode` :
  i. `r` (*read*) : accès en **lecture uniquement** et **le pointeur de fichier sera au début du fichier**; si le fichier **n'existe pas** il y aura une **erreur** (*par défaut*)
  ii. `w` (*write*) : accès en **écriture uniquement** et **le pointeur de fichier sera au début du fichier**; si le fichier **existe** alors il sera **écrasé**, sinon il sera **créé**.
  iii. `a` (*append*) : *idem* que `w` mais **le pointeur de fichier sera placé en fin du fichier** (*si celui-ci existe déjà*)
  iv. `b` (*binary*) : **ajouté à l'un des trois autres modes** et désigne **lecture/écriture/ajout** d'un fichier en **mode binaire**

2. faire des **opérations de lecture/écriture**
3. **fermer le fichier** avec `<file>.close()` (`<file>` est un **descripteur de fichier** obtenu via la **méthode** `open()` de **type** `_io.TextIOWrapper`)

Lecture/Écriture d'un fichier

1. `<file>.read() -> str` : **lire** le **contenu** d'un **fichier entièrement** (`mode = 'r'`) et le **renvoyer** en tant que *string*
2. `<file>.write(chaine) -> int` :
   - **écrire** un *string* `chaine` dans un **fichier** `file` (`mode = 'w'` ou `mode = 'a'`)
   - **renvoyer** le **nombre de caractères écrits** dans le fichier
3. `<file>.readlines() -> list of strings` : **lire** le contenu d'un **fichier**, stocker chaque **ligne** dans un *string* et renvoyer les lignes dans une **liste de strings**

Le mot-clé `with`

1. **problème** : lors des **E/S** on peut avoir des **erreurs/exceptions** qui peuvent **arrêter l'exécution** du **script** sans avoir

fermé les **fichiers**

2. **solution** :
   - **utilisation** d'un **gestionnaire de contexte** qui sera chargé de l'**ouverture et de la fermeture d'un fichier dans un bloc de code** *même si des erreurs se produisent pendant l'exécution du bloc*
   - **définition** de ce **gestionnaire de contexte** via l'instruction utilisant le **mot-clé** `with`
3. **syntaxe** :

```
with expression [as target] :
  #instructions
```

remarque : possibilité de **vérifier** la **fermeture d'un fichier** `target` en affichant la valeur de `target.closed` qui vaudra :

- `True` *si* le **fichier est fermé**
- `False` *sinon*

Sérialisation et désérialisation

1. **définition** : **stocker/récupérer** des **objets** dans/depuis des **fichiers** en **format binaire**, de manière à ce que ce fichier ne soit pas **lisible à l'oeil humaine** et **utilisable par d'autres programmes désérialisant le fichier contenant les objets sérialisés**
2. en **Python** : via le **module** `pickle`

Le module `pickle`

1. non importé automatiquement par l'interprète : `import pickle`
2. **description** : un **module** utilisé pour la **sérialisation/désérialisation des objets**

Classes

1. `Pickler` :
   - **description** : classe définissant un *Pickler* utilisé pour la **sérialisation d'objets**
   - **constructeur** : `Pickler(<file>)` : définit un **sérialiseur** (*Pickler*) permettant de **sérialiser** des **objets** dans le **fichier** `file`
   - **méthodes** :
     - `<serialiser>.dump(objet)` : **sérialiser l'objet** `objet` via le **sérialiseur** `serialiser` défini sur le fichier `file`
2. `Unpickler` :
   - **description** : classe définissant un *Unpickler* utilisé pour la **désérialisation d'objets**
   - **constructeur** : `Unpickler(<file>)` : définit un **désérialiseur** (*Unpickler*) permettant de **désérialiser** des **objets sérialisés** au préalable dans le **fichier** `file`
   - **méthodes** :
     - `<deserialiser>.load()` : **désérialiser** le **dernier objet sérialisé** dans le **fichier** `file` sur lequel est défini le **désérialiseur** `deserialiser`

Exemples

```
#lecture du contenu d'un fichier "fichier.txt"
#dans le répertoire de travail courant
mon_fichier = open("fichier.txt", "r")
print(mon_fichier.read())
mon_fichier.close()
#####################################
```

```
#écriture dans un fichier "fichier.txt" dans le répertoire de travail courant
mon_fichier = open("fichier.txt", "w")
mon_fichier.write("Premier test d'écriture dans un fichier via Python")
mon_fichier.close()
#####################################

#lecture du contenu d'un fichier "fichier.txt"
#dans le répertoire courant en utilisant l'instruction utilisant "with"
with open("fichier.txt", "r") as mon_fichier:
    print(mon_fichier.read())
#####################################

#sérialisation d'un dictionnaire de scores dans un fichier "donnee"
#dans le répertoire de travail courant en utilisant le module "pickle"
from pickle import Pickler

score = {
  "joueur 1": 5,
  "joueur 1": 5,
  "joueur 2": 35,
  "joueur 3": 20,
  "joueur 4": 2
}

with open("donnees", "wb") as mon_fichier:
  mon_pickler = Pickler(mon_fichier)
  mon_pickler.dump(score)
#####################################

#désérialisation d'un dictionnaire de scores sérialisé dans un fichier "donnee"
#dans le répertoire de travail courant en utilisant le module "pickle"
from pickle import Unpickler

with open("donnees", "rb") as mon_fichier:
  mon_unpickler = Unpickler(mon_fichier)
  score_recupere = mon_unpickler.load()
  print(score_recupere)
```

Portée des variables et références

Portée des variables

1. tous les **objets** définis en **Python** en dehors de toute **fonction**, sont définis dans l'**espace globale**
2. toute **fonction** définie dans l'**espace globale** :
   - a son **propre espace de noms** ayant sa **propre portée**; on parle ainsi d'**espace local d'une fonction**
   - tous les **objets définis dans l'espace local** d'une **fonction** seront **détruits en sortant du bloc de celle-ci**
3. une **fonction définie** dans l'**espace globale** aura accès à un **objet global** :
   - **en lecture** :

     ```
     obj_global = <valeur>
     def fonction([params]):
       #instructions accédant à l'objet global "obj_global" en lecture uniquement
     ```

   - **en écriture**, seulement si l'on **déclare l'objet global au tout début du corps de la fonction** (*après sa docstring*) en tant que `global` :

     ```
     obj_global = <valeur>
     ```

```
def fonction([params]):
  global obj_global
  #instructions accédant à l'objet global "obj_global" en lecture et écriture
```

4. méchanisme de **recherche** d'un **objet** utilisé dans une **fonction** :
   - lors de l'évaluation d'une variable désignant un objet dans une fonction, l'interprète **Python** commence à rechercher l'objet depuis l'**espace local** associé à la fonction
   - s'il n'en trouve pas, il recherche dans l'**espace depuis lequel a été appelée la fonction**,
   - ...
   - s'il n'en trouve pas, il recherche dans l'**espace globale**
   - s'il n'en trouve pas, il lève une exception de type `NameError`

Les références

Rappel : **tout est objet en Python**

```
#définition d'un objet o1
#(o1 est de type non primitif
#(\notin (entiers, flottants, booléens, chaînes de caractères)))
o1 = <valeur>

#copie superficielle de o1 dans o2 via l'opérateur d'affectation.
#o1 et o2 sont deux références pointant sur le même objet en mémoire
o1 = o2 :
```

1. tout argument de type **objet** (*sauf les types primitifs*) est passé **en référence** à une **fonction** admettant un **paramètre objet**. Autrement dit, cette **fonction** pourra **modifier l'objet** (*s'il n'est pas immuable*) via ses **méthodes** (*et non pas via l'opérateur d'affectation*)
2. une **fonction** ne peut modifier **aucun objet** passé en **argument** via l'**opérateur d'affectation** :
   - l'**affectation** d'une **nouvelle valeur** à un **paramètre objet** dans le **corps** de la **fonction** créera un **nouvel objet** dans l'**espace local de la fonction**
   - le **nom** du **nouvel objet masquera** celui de l'**objet passé en paramètre** -> la **fonction** n'aura **plus accès à l'objet passé en argument**
3. pour créer une **copie profonde d'un objet** (*deux objets différents en mémoire*), il suffit d'utiliser le **constructeur de la classe** définissant l'**objet** que l'on souhaite **copier** : `obj_copie = <type>(obj)`

Exemples

```
#Exemple d'affichage d'une variable globale depuis l'espace local d'une fonction
a = 5
def print_a():
    print("la variable a =", a)

print_a() #la variable a = 5
a = 8
print_a() #la variable a = 8
################################################

#Exemple d'une fonction modifiant un objet transmis en paramètre
#via une méthode sur l'objet appelé (non pas via affectation)
def ajouter(liste, valeur):
    """Fonction insérant à la fin de la liste la valeur à ajouter"""
    liste.append(valeur)

ma_liste = ["a", "e", "i"]
print(ma_liste) #["a", "e", "i"]
```

```
ajouter(ma_liste, "o")
print(ma_liste) #["a", "e", "i", "o"]
##################################################

#Exemple d'une copie superficielle de listes
#(deux références pointant sur le même objet en mémoire)
#pour décrire le mécanisme des références
l1 = [1, 2, 3]
l2 = l1

print("l1 = {} d'identifiant {}".format(l1, id(l1))) #l1 = [1, 2, 3] d'identifiant 140311970066504
print("l2 = {} d'identifiant {}".format(l2, id(l2))) #l2 = [1, 2, 3] d'identifiant 140311970066504

print("l1 == l2 ? {}".format(l1 == l2)) #l1 == l2 ? True
print("l1 is l2 ? {}".format(l1 is l2)) #l1 is l2 ? True

l1.append(4)
print("l1 = {} d'identifiant {}".format(l1, id(l1))) #l1 = [1, 2, 3, 4] d'identifiant 140311970066504
print("l2 = {} d'identifiant {}".format(l2, id(l2))) #l2 = [1, 2, 3, 4] d'identifiant 140311970066504

print("l1 == l2 ? {}".format(l1 == l2)) #l1 == l2 ? True
print("l1 is l2 ? {}".format(l1 is l2)) #l1 is l2 ? True
##################################################

#Exemple d'une copie profonde de listes
#(en utilisant le constructeur de la classe des listes "list")
l1 = [1, 2, 3]
l2 = list(l1)

print("l1 = {} d'identifiant {}".format(l1, id(l1))) #l1 = [1, 2, 3] d'identifiant 140011546239048
print("l2 = {} d'identifiant {}".format(l2, id(l2))) #l2 = [1, 2, 3] d'identifiant 140011546240264

print("l1 == l2 ? {}".format(l1 == l2)) #l1 == l2 ? True
print("l1 is l2 ? {}".format(l1 is l2)) #l1 is l2 ? False

l1.append(4)
print("l1 = {} d'identifiant {}".format(l1, id(l1))) #l1 = [1, 2, 3, 4] d'identifiant 140011546239048
print("l2 = {} d'identifiant {}".format(l2, id(l2))) #l2 = [1, 2, 3] d'identifiant 140011546240264

print("l1 == l2 ? {}".format(l1 == l2)) #l1 == l2 ? False
print("l1 is l2 ? {}".format(l1 is l2)) #l1 is l2 ? False
##################################################

#Exemple d'une modification d'une variable globale depuis une fonction
def increment_i():
    """Fonction permettant d'incrémenter la variable globale i"""
    global i;
    i += 1

i = 4
print(i) #4

increment_i()
print(i) #5
```

TP : Un bon vieux pendu

```
#donnees.py
"""Module utilitaire contenant les données qui seront utilisées par les fonctions mettant en oeuvre le jeu
SCORES_FILE_NAME = "scores"
MAX_TOURS = 10
MAX_PSEUDO_LENGTH = 10
```

```
SECRETS = [
"dude",
"leave",
"crap",
"alone",
"perhaps",
"diesel",
"elephant",
"andorra",
"belgium"
]

#fonctions.py
"""Module contenant les fonctions utilitaires utilisées pour la création du jeu de pendu"""
from donnees import *
import random
import pickle

def get_name():
    """Demande le nom de l'utilisateur et le renvoie"""
    name = input("Quel est votre nom ? ")
    return name if len(name) <= MAX_PSEUDO_LENGTH else name[:MAX_PSEUDO_LENGTH]

def get_score(player):
    """Recherche la liste des scores
    si player est dans la liste, alors renvoie le score associé
    sinon renvoie 0"""

    score = 0

    try:
        scores_file = open(SCORES_FILE_NAME, "rb")
        unpickler = pickle.Unpickler(scores_file)
        scores = unpickler.load()
        print("Scores : {}".format(scores))
        if player in scores.keys():
            score = scores[player]
    except EOFError:
        pass
    finally:
        scores_file.close()

    return score

def get_rand_secret():
    """Cherche un mot secret parmi la liste des mots secrets et le renvoie"""
    return random.choice(SECRETS)

def set_guess(secret_length):
    """Prepare le mot de devination qui sera utilisé par l'utilisateur
    au fur et à mesure du jeu et le renvoie"""
    i = 0
    guess = ""
    while i < secret_length:
        guess += "*"
        i += 1
    return guess

def init_jeu():
    """Fonction initialisant le jeu en :
    1. demandant au joueur son nom et récupérant son score s'il existe ou l'initialiser sinon
    2. préparer un mot secret parmi une liste de mots secrets et le mot de devination utilisé par le joueur
    """
    player = get_name()
    score = get_score(player)
    secret = get_rand_secret()
    guess = set_guess(len(secret))
```

```
        return player, score, secret, guess

    def get_lettre():
        """Demander à l'utilisateur de saisir une lettre afin de deviner le mot secret"""
        lettre = ""
        while len(lettre) < 1 or not lettre.isalpha():
            lettre = input("Choisissez une lettre : ")
        return lettre

    def remplacer_etoile_par_lettre(secret, lettre, guess):
        result = str(guess)
        for i, char in enumerate(secret):
            if char == lettre:
                result = result[:i] + lettre + result[(i+1):]
        return result

    def sauvegarder_score(player, score):
        """Sauvegarder le score du joueur dans scores"""
        scores = {}

        try:
            scores_file = open(SCORES_FILE_NAME, "rb")
            unpickler = pickle.Unpickler(scores_file)
            scores = unpickler.load()
        except EOFError:
            print("Le fichier de scores est vide !")
        finally:
            scores_file.close()

        scores[player] = score

        with open(SCORES_FILE_NAME, "wb") as scores_file:
            pickler = pickle.Pickler(scores_file)
            pickler.dump(scores)

    def jouer_partie():
        """Jouer une partie du jeu pendu"""
        tour = 0
        discovered = False
        player, score, secret, guess = init_jeu()
        while tour < MAX_TOURS and discovered is False :
            print("Le mot secret : {}.\nEssayez de le deviner ! (tours : {}; score : {})".format(guess, MAX_TOU
            lettre = get_lettre()
            if lettre in secret:
                guess = remplacer_etoile_par_lettre(secret, lettre, guess)
                print("guess = {}".format(guess))
                print("Bravo !")
                print("================")
            else:
                print("Raté !")
                print("================")
            tour += 1
            if guess.count("*") == 0:
                discovered = True

        if discovered is True:
            print("Vous avez gagné !")
            score += MAX_TOURS - tour
        else:
            print("Vous avez perdu ! Le mot secret était {}".format(secret))

        print("Votre score est : {}".format(score))
        sauvegarder_score(player, score)

    def jouer():
        """Jouer le jeu du pendu"""
        nouvelle_partie = True
```

```
    while nouvelle_partie is True:
        jouer_partie()
        choix = input("Voulez-vous jouer une autre partie (O/N) ? ")
        choix = choix[0]
        if choix != "O":
            nouvelle_partie = False

#pendu.py
from fonctions import jouer
jouer()
```

La programmation orientée objet

Les classes

Rappel

1. **objet** : une **structure de données composée** :
   - d'une **partie structurale statique** (les **attributs** sous forme de **variables**) ;
   - d'une **partie comportomentale dynamique** (les **méthodes** sous forme de **fonctions**)
2. **classe** : un **type de données** désignant un **modèle** permettant d'**instancier des objets**
3. **classe singleton** : une **classe** qui ne peut être **instanciée** qu'**une seule fois**

Syntaxe

```
class NomClasse:
  """docstring de la classe"""

  #class attributes
  attribut_classe = valeur_par_defaut
  ...

  #constructor
  def __init__(self[, a1[, ...]]) : #self \equiv this sous C++ et Java
    self.a1 = a1 | valeur_par_defaut
    ...

  #methods
  ##instance methods
  #Les méthodes d'instance agissent sur l'objet courant via self
  def methode_instance1(self[a1[, ...]]):
    #code

  ...

  ##class methods
  #Les méthodes de classe agissent sur des attributs de classes via cls
  def methode_classe1(cls[, a1[, ...]]):
    #code
    #On peut appeler les méthodes de classe (et statiques) de deux manières :
      #1. NomClasse.methode_classe1([...])
      #2. objet = NomClasse([...])
      #   objet.methode_classe1([...])
```

```
    ...

    ##static methods
    #Les méthodes statiques sont un peu comme les méthodes de classe
    #sauf qu'elles ne prennent aucun paramètre
    def methode_statique1():
      #code

    ...

    #class methods declarations
    methode_classe1 = classmethod(methode_classe1)

    #static methods declarations
    methode_statique1 = staticmethod(methode_statique1)
    ...

#instanciation d'un objet "objet" de la classe "NomClasse" en l'initialisant
#via le constructeur [en passant les arguments a1[, ...]]
objet = NomClasse([a1[, ...]])
```

Introspection

Définition : des **méchanismes** permettant d'**exploiter un objet** : connaître ses **méthodes** ou **attributs**

La fonction `dir()` et l'attribut `__dict__`

1. `dir(objet) -> list of strings` : renvoie la **liste** des **attributs/méthodes** (**héritées et définies**) de l'**objet** `objet` **passé en paramètre** quel que soit son **type** (*methode, fonction, classe, module, ...*)
2. `<objet>.__dict__` :
   - renvoie un **dictionnaire** contenant les **attributs** de l'**objet** `objet` comme **clés** et les **valeurs** correspondant à ses **attributs** comme **valeurs**
   - **remarque** : on peut **modifier** les **valeurs** des **attributs** d'un **objet** à travers l'**attribut spécial** `__dict__` :
     `objet.__dict__["attribut"] = nouvelle_valeur`

Remarque

Il faut **privilégier** l'**utilisation** de l'**introspection uniquement** quand on aura vraiment besoin

Exemples

```
#Exemple d'une classe Personne
class Personne:
    """Classe définissant une personne caractérisée par :
    - son nom
    - son prenom
    - son âge
    - son lieu de résidence"""

    #class attribute
    personnes_creees = 0

    #constructor
    def __init__(self, nom, prenom, age, lieu_residence):
        """Constructeur de la classe Personne initializant ses attributs"""
        self.nom = nom
        self.prenom = prenom
        self.age = age
        self.lieu_residence = lieu_residence
        self.telephone = "N/A"
```

```
        Personne.personnes_creees += 1

print("Personnes créées = {}".format(Personne.personnes_creees)) #Personnes créées = 0

bertrand = Personne("bertrand", "benjamin", 45, "Paris")
print(bertrand) #<__main__.Personne object at 0x763fb39b0f0
print("bertrand.prenom = {}".format(bertrand.prenom)) #bertrand.prenom = benjamin
print("bertrand.nom = {}".format(bertrand.nom)) #bertrand.nom = bertrand
print("bertrand.age = {}".format(bertrand.age)) #bertrand.age = 45
print("bertrand.lieu_residence = {}".format(bertrand.lieu_residence)) #bertrand.lieu_residence = Paris
print("bertrand.telephone = {}".format(bertrand.telephone)) #bertrand.telephone = N/A

bertrand.telephone = "0612233445"
print("bertrand.telephone = {}".format(bertrand.telephone)) #bertrand.telephone = 0612233445
print("Personnes créées = {}".format(Personne.personnes_creees)) #Personnes créées = 1
############################################################

#Exemple d'une classe Compteur
class Compteur:
    """Classe possédant un attribut de classe s'incrémentant
    à chaque fois que l'on crée un objet"""

    #class attributes
    objets_crees = 0

    #constructor
    def __init__(self):
        Compteur.objets_crees += 1

    #methods
    ##class methods
    def combien(cls):
        print("Objets crées jusqu'à présent : {}".format(cls.objets_crees))

    #class methods declarations
    combien = classmethod(combien)

Compteur.combien() #Objets crées jusqu'à présent : 0

a = Compteur()
Compteur.combien() #Objets crées jusqu'à présent : 1

b = Compteur()
Compteur.combien() #Objets crées jusqu'à présent : 2
############################################################

#classe TableauNoir
class TableauNoir:
    """Une surface sur laquelle on peut écrire
    que l'on peut lire et effacer par jeu de méthodes."""

    #constructor
    def __init__(self):
        """Par défaut, notre surface est vide"""
        self.surface = ""

    #methods
    ##instance methods
    def ecrire(self, message):
        """Écrire sur la surface du tableau.
        Si la surface n'est pas vide, on saute une ligne
        avant de rajouter le message à écrire"""

        if self.surface != "":
            self.surface += "\n"
        self.surface += message
```

```
    def lire(self):
        """Lire le contenu de la surface du TableauNoir courant"""
        print("Surface = {}".format(self.surface))

    def effacer(self):
        """Effacer le contenu de la surface du TableauNoir courant"""
        self.surface = ""

tab = TableauNoir()
tab.lire() #Surface =

tab.ecrire("Coool ! Les vacances sont là !")
tab.lire() #Surface = Coool ! Les vacances sont là !

tab.ecrire("Joyeux Noël !")
tab.lire() #Surface = Coool ! Les vacances sont là !\n Joyeux Noël !

tab.effacer()
tab.lire() #Surface =
################################################################

#classe Test utilisée pour illustrer l'utilisation de la fonction dir()
#et de l'attribut spécial __dict__
class Test:

    #constructor
    def __init__(self):
        self.attribut = "ok"

    #methods
    #instance methods
    def afficher_attribut(self):
        print("Mon attribut est {}".format(self.attribut))

test = Test()
test.afficher_attribut() #Mon attribut est ok
print(dir(test)) #liste des attributs et méthodes prédéfinies pour l'objet test

test.__dict__["attribut"] = "plus ok"
test.afficher_attribut() #Mon attribut est plus ok
```

Les propriétés

Introduction

1. **principe de l'encapsulation** : cacher les **attributs** et ne pas permettre l'**accès direct** à leurs valeurs pour **sécuriser l'objet** et **contrôler** la manière et les conditions permettant de le **mettre à jour** et de **changer son état**
2. **accesseurs et mutateurs** :
   - **méthodes** permettant respectivement d'**accéder aux attributs de l'objet en lecture et écriture**
   - **accesseurs** : `_get_attribut()` ; **mutateurs** : `_set_attribut(valeur)`
3. en **Python** : **tout est publique**, il n'y a pas la notion de *private* qu'on trouvera ailleurs (*C++, Java*). Pour **contrôler l'accès aux attributs** on utilise les **propriétés**

Principe

1. les **propriétés** sont des **objets particuliers du langage Python** permettant d'**abstraire le contrôle d'accès à un attribut** pour permettre son accès via la **même syntaxe** d'une manière transparente quelles que soient les **permissions d'accès**
2. les **propriétés** permettent de :

- rendre un **attribut complétement inaccessible** depuis l'**extérieur** ou **accessible en lecture uniquement**
- **mettre à jour un attribut** lors de l'**accès** à un **autre attribut**
- ...

3. les **propriétés** sont définies lors de la **conception de la classe** dans le **corps de cette dernière** via un **objet** de la **classe** `property` dont le **constructeur** prend les **paramètres** suivants :
   - un **paramètre** désignant un **accesseur** appelé lors de l'**accès à l'attribut en lecture**
   - un **paramètre** désignant un **mutateur** appelé lors de l'**accès à l'attribut en écriture**
   - un **paramètre** désignant une **méthode** appelée lors de la **destruction de l'attribut** (`del objet.attribut`)
   - un **paramètre** désignant une **méthode** appelée lors du besoin de l'**aide sur l'attribut** (`help(objet.attribut)`)

Exemples

```
#définition d'une propriété pour l'attribut lieu_residence
#de la classe Personne définie précédemment
class Personne:

    #constructor
    def __init__(self, nom, prenom):
        self.nom = nom
        self.prenom = prenom
        self.age = 33
        self._lieu_residence = "Paris"
        #le nom d'un attribut pour lequel on souhaite définir une propriété
        #commence par un "_" pour indiquer que l'on n'y accède pas
        #directement via la notation objet.attribut (par convention)

    #methods
    #instance methods
    def _get_lieu_residence(self):
        #le nom d'une méthode qui sera utilisée pour définir une propriété
        #commence par un "_" pour indiquer que l'on n'y accède pas
        #directement via la notation objet.methode() (par convention)
        print("On accède à l'attribut lieu_residence !!")
        return self._lieu_residence

    def _set_lieu_residence(self, nouvelle_residence):
        print("Attention, il semble que {} déménage à {}".format(self.prenom, \
        nouvelle_residence))
        self._lieu_residence = nouvelle_residence

    #attribute-property binding
    #création de la propriété lieu_residence qui remplacera l'attribut
    #_lieu_residence lors de son accès
    lieu_residence = property(_get_lieu_residence, _set_lieu_residence)

jean = Personne("Micado", "Jean")
print("Nom : {}".format(jean.nom)) #Nom : Micado
print("Age : {}".format(jean.age)) #Age : 33
print("Lieu de Résidence : {}".format(jean.lieu_residence)) #On accède à l'attribut lieu_résidence !\nLieu

jean.lieu_residence = "Berlin" #Attention, il semble que Jean déménage à Berlin
print("Lieu de Résidence : {}".format(jean.lieu_residence)) #On accède à l'attribut lieu_résidence !\nLieu
```

Les méthodes spéciales (*magic methods*)

Introduction

1. **définition** : les **méthodes spéciales** sont des **méthodes d'instance reconnues par Python** en tant que méthodes

utilisables **dans des contextes particuliers** (*e.g. lors de l'appel de certains opérateurs, ...*)

2. **syntaxe** : `__methodespeciale__()`
3. **utilités** :
   - **contrôler l'état d'un objet** : *création, accès et suppression*
   - **surcharger les opérateurs pour leur utilisation sur un objet défini** (*méthodes de conteneurs, méthodes mathématiques, méthodes de comparaison, ...*)
4. **remarques** :
   - il existe **plusieurs méthodes spéciales prédéfinies** pour un **objet** lors de sa **création** dont la plupart sont **définies** par la **classe** `object` .
   - si le **comportement associé à une méthode spéciale n'est pas spécialisé dans la classe**, **Python** utilisera un **comportement par défaut** qui lui est défini par la **classe** `object`

Méthodes spéciales pour l'édition de l'objet

1. ces **méthodes** seront appelées lors de la **création/suppression** de l'**objet**
2. le **constructeur** :
   - `__init__(self[, args[, ...]])` :
   - appelée lors de la **création de l'objet**
3. le **destructeur** :
   - `__del__(self)`
   - appelée lors de la **suppression de l'objet** (`del objet` ou *destruction de l'espace de noms contenant l'objet*)

Méthodes spéciales pour la représentation de l'objet

1. la **méthode** `__repr__()` :
   - `__repr__(self)`
   - appelée lors de l'**affichage formel de l'objet**
   - appelée sur un **objet** aussi via la **fonction built-in** `repr(objet)`
2. la **méthode** `__str__()` :
   - `__str__(self)`
   - appelée :
     a. lors de **l'affichage informel de l'objet** via la **fonction** `print()`
     b. lors de la **conversion de l'objet en string** (via `str(objet)`)

- si cette méthode n'est pas définie, **Python** appellera la **méthode** `__repr__()`

Méthodes spéciales pour l'accès aux attributs de l'objet

1. la **méthode** `__getattr__()` :
   - `__getattr__(self, attribut) -> None`
   - appelée lorsqu'on essaye d'**accéder en lecture à un attribut** `attribut` **ne figurant pas dans l'objet** et qui sera passé en **paramètre** de cette méthode
2. la **méthode** `__setattr__()` :
   - `__setattr(self, attribut, valeur) -> None`
   - appelée lorsqu'on essaye d'**accéder en écriture à un attribut** `attribut` en lui **affectant la valeur** `valeur`
   - **remarque** : on ne peut pas **modifier un attribut dans cette méthode spéciale** :
     a. si on essaie d'**accéder à un attribut en écriture**, cette méthode sera appelée, et si l'on essaie d'**accéder en**

écriture à un autre attribut dans le corps de cette méthode, elle sera encore **rappelée**, et donc on risque de tomber dans une **boucle infinie**

b. pour **accéder à un attribut en écriture dans le corps de la méthode** `__setattr__()` spécialisée, on utilise la **méthode** `__setattr__()` de la **classe** `object` :

```
object.__setattr__(self, attribut, valeur)
```

3. la **méthode** `__delattr__()` :
   - `__delattr__(self, attribut)`
   - appelée lorsqu'on essaye de **supprimer un attribut** `attribut` de **l'objet courant**
   - `AttributeError` : exception levée en cas d'**accès à un attribut indéfini pour un objet**
   - si on souhaite **supprimer** un **attribut** d'un **objet** dans le **corps** de cette **méthode** il faut le faire via la **méthode** `__delattr__()` de la **classe** `object` (*même raisonnement que celui de* `__setattr__()`)
4. **fonctions built-in semblables** :
   - `getattr(objet, attribut)` : renvoie la **valeur** de l'**attribut** `attribut` de l'**objet** `objet` (*equiv* `objet.attribut`)
   - `setattr(objet, attribut, valeur)` : **affecter** la **valeur** `valeur` à l'**attribut** `attribut` de l'**objet** `objet` (*equiv* `objet.attribut = valeur`)
   - `delattr(objet, attribut)` : **supprimer** l'**attribut** `attribut` de l'**objet** `objet` (*equiv* `del objet.attribut`)
   - `hasattr(objet, attribut)` : **renvoie** `True` si l'**objet** `objet` **possède l'attribut** `attribut` , `False` sinon

Méthodes spéciales de conteneurs

1. ces méthodes spéciales permettent la **surchage d'opérateurs utilisables avec les conteneurs**
2. **surcharge d'opérateurs** : **redéfinir** des **opérateurs** pour des **classes créées**

Accès aux éléments d'un conteneur

1. **conteneurs** = *strings, listes, dictionnaires, ...*
2. la **méthode** `__getitem__()` :
   - `__getitem__(self, index)`
   - appelée lorsqu'on essaie d'**accéder en lecture** à un **item** de notre **objet conteneur** via l'**opérateur** `[]` : `objet[index]`
   - utilisée pour **surcharger l'opérateur** `[]` **en lecture**
3. la **méthode** `__setitem__()` :
   - `__setitem__(self, index, valeur)`
   - appelée lorsqu'on essaie d'**accéder en écriture** à un **item** de notre **objet conteneur** via l'**opérateur** `[]` : `objet[index] = valeur`
   - utilisée pour **surcharger l'opérateur** `[]` **en écriture**
4. la **méthode** `__delitem__()` :
   - `__delitem__(self, index)`
   - appelée lorsqu'on essaie de **supprimer** un **item** de notre **objet conteneur** : `del objet[index]`
   - utilisée pour **surcharger l'opérateur** `[]` **en suppression**

La méthode spéciale derrière le mot-clé `in`

1. la **méthode** `__contains__()` :
   - `__contains__(self, element)`
   - appelée lorsqu'on essaie de **vérifier** si un **élément** se trouve dans notre **objet conteneur** via le **mot-clé** `in`

- `element in conteneur` *equiv* `conteneur.__contains__(element)`
- utilisée pour **surcharger l'utilisation du mot-clé** `in` pour vérifier l'**appartenance** d'un **item** à notre **objet conteneur**

2. la **méthode** `__len__()` :
   - `__len__(self)`
   - appelée lorsqu'on essaye d'appeler la **fonction** `len()` sur notre **objet**
   - `len(conteneur)` *equiv* `conteneur.__len__()`
   - utilisée pour **surcharger l'utilisation** de la **fonction** `len()` pour vérifier la **taille** de notre **objet conteneur**

Méthodes spéciales mathématiques

1. Ces méthodes spéciales permettent la **surcharge d'opérateurs arithmétiques** (`+ - * /`)
2. la **méthode spéciale** `__add__()` :
   - `__add__(self, objet_a_ajouter)`
   - appelée lorsqu'on essaye d'appeler l'**opérateur** `+` : `objet + objet_a_ajouter`
   - utilisée pour **surcharger l'opérateur** `+`
   - `objet + objet_a_ajouter` *equiv* `objet.__add__(objet_a_ajouter)`
3. d'autres **méthodes spéciales mathématiques** :
   - `__sub__()` : **surcharge de l'opérateur** `-`
   - `__mul__()` : **surcharge de l'opérateur** `*`
   - `__truediv__()` : **surcharge de l'opérateur** `/`
   - `__floordiv__()` : **surcharge de l'opérateur** `//`
   - `__mod__()` : **surcharge de l'opérateur** `%`
   - `__pow__()` : **surcharge de l'opérateur** `**`
   - `__r<methode>__()` : **surcharger l'associativité droite** de l'**opérateur arithmétique** associé à la **méthode spéciale** `methode` (`__radd__()` , `__rsub__()` , `__rmul__()` , ...)
   - `__i<methode>__()` : **surcharger l'opérateur raccourci** associé à l'**opérateur arithmétique** associé à la **méthode spéciale** `methode` (`__iadd__()` , `__isub__()` , `__imul__()` , ...)

Méthodes de comparaison

1. Ces **méthodes spéciales** permettent de **surcharger des opérateurs de comparaisons** (`==` , `!=` , `<` , `<=` , `>` , `>=`)
2. la **méthode spéciale** `__eq__()` :
   - `__eq__(self, objet_a_comparer)`
   - appelée lorsqu'on essaye d'appeler l'**opérateur** `==` : `objet == objet_a_comparer`
   - utilisée pour **surcharger l'opérateur** `==`
   - `objet == objet_a_ajouter` *equiv* `objet.__eq__(objet_a_comparer)`
3. d'autres **méthodes spéciales de comparaison** :
   - `__ne__()` : **surcharge de l'opérateur** `!=`
   - `__lt__()` : **surcharge de l'opérateur** `<`
   - `__le__()` : **surcharge de l'opérateur** `<=`
   - `__gt__()` : **surcharge de l'opérateur** `>`
   - `__ge__()` : **surcharge de l'opérateur** `>=`
4. **remarque** : si **Python** n'arrive pas à faire l'une des **opérations** par l'un des **opérateurs de comparaison**, il essaiera de faire l'**opération inverse aussi** : *e.g.* `o1 < o2` **ne marche pas**, alors **Python** essaiera `o2 >= o1`

Methodes spéciales utiles à pickle

1. la **méthode spéciale** du **module** `pickle` `__getstate__()` :
   - `__getstate__(self)`
   - appelée **avant la sérialisation d'un objet** (*écriture dans un fichier*) et son **résultat** est **sérialisé** par `pickle`
   - **par défaut**, `pickle` **enregistre les attributs** de l'**objet à sérialiser** dans un **dictionnaire** qui sera ensuite **sérialisé** (le **dictionnaire** est **contenu** dans `objet.__dict__`)
2. la **méthode spéciale** du **module** `pickle` `__setstate__()` :
   - `__setstate__(self, objet_recupere)`
   - appelée **après la désérialisation d'un objet**, tel que :
     a. l'**objet** `Unpickler` **lit le fichier** contenant l'**objet sérialisé** et **récupère** le **dictionnaire** contenant ses **attributs** et leurs **valeurs** (*renvoyé (ou pas) par la méthode* `__getstate__()`)
     b. *si* la **méthode** `__setstate__()` est **définie**, le **dictionnaire** lui est **retourné** et elle peut ainsi le **modifier avant de le renvoyer** pour qu'il sera utilisé à **construire l'objet souhaité** en l'**écrivant** dans la **propriété** `__dict__` de ce dernier
     c. *sinon* (*par défaut*), le **dictionnaire récupéré** par `Unpickler` est **directement écrit** dans `__dict__` de l'**objet à construire** par **désérialisation**
3. **remarque** : la **méthode** `__getstate__()` peut ne pas **envoyer** un **dictionnaire d'attributs** désignant l'**objet à sérialiser**, mais dans ce cas il faut impérativement **définir** la **méthode** `__setstate__()` pour **savoir désérialiser** (*Python ne saura pas comment le faire si* `__getstate__()` *n'envoie pas un dictionnaire*)

Exemples

```
#classe Personne spécialisant les méthodes spéciales
#__init__(), __del__(), __repr__(), __str__()
class Personne:

    #constructor
    def __init__(self, nom, prenom):
        self.nom = nom
        self.prenom = prenom
        self.age = 33
        self._lieu_residence = "Paris"

    #destructor
    def __del__(self):
        print("C'est la fin !")

    #methods
    ##instance methods
    def _get_lieu_residence(self):
        print("On accède à l'attribut lieu_residence !!")
        return self._lieu_residence

    def _set_lieu_residence(self, nouvelle_residence):
        print("Attention, il semble que {} déménage à {}".format(self.prenom, \
        nouvelle_residence))
        self._lieu_residence = nouvelle_residence

    ##magic methods
    #représentation formelle d'une personne
    def __repr__(self):
        return "Personne : {} {} ({} ans), {}".format(self.nom.upper(), \
        self.prenom, self.age, self.lieu_residence)

    #représentation informelle d'une personne (\equiv toString() sous Java)
    def __str__(self):
        return "{} {}, âgé de {} et vivant en {}".format(self.prenom, \
        self.nom, self.age, self.lieu_residence)
```

```
    ###attribute-property binding
    lieu_residence = property(_get_lieu_residence, _set_lieu_residence)
    #création de la propriété lieu_residence qui remplacera
    #l'attribut _lieu_residence lors de son accès

jean = Personne("Micado", "Jean")
print(jean)
#Personne : MICADO Jean (33 ans), Paris (si __str__ est indéfinie) |
#Jean Micado, âgé de 33 et vivant en Paris (sinon)

print("Nom : {}".format(jean.nom)) #Nom : Micado
print("Age : {}".format(jean.age)) #Age : 33
print("Lieu de Résidence : {}".format(jean.lieu_residence)) #On accède à l'attribut lieu_résidence !\nLieu

jean.lieu_residence = "Berlin" #Attention, il semble que Jean déménage à Berlin
print("Lieu de Résidence : {}".format(jean.lieu_residence)) #On accède à l'attribut lieu_résidence !\nLieu

print(jean)
#Personne : MICADO Jean (33 ans), Berlin (si __str__ est indéfinie) |
#Jean Micado, âgé de 33 et vivant en Berlin (sinon)
#C'est la fin !
################################################################

#classe Protege spécialisant les méthodes spéciales
#__getattr__(), __setattr__(), __delattr__()
class Protege:

    #constructor
    def __init__(self):
        self.a = 1
        self.b = 2
        self.c = 3

    #methods
    ##magic methods
    def __getattr__(self, attribut):
        print("Alerte ! Il n'y a pas d'attribut {} ici !".format(attribut))

    def __setattr__(self, attribut, valeur):
        object.__setattr__(self, attribut, valeur)

    def __delattr__(self, attribut):
        raise AttributeError("Vous ne pouvez supprimer aucun attribut " \
        + "de cette classe !")

pro = Protege()
print("pro.a = {}".format(pro.a)) #pro.a = 1
print("pro.b = {}".format(pro.b)) #pro.b = 2
print("pro.e = {}".format(pro.e)) #Alerte ! Il n'y a pas d'attribut e ici !\npro.e = None

del pro.a
#AttributeError exception levée avec le message
#"Vous ne pouvez supprimer aucun attribut de cette classe !"
################################################################

#classe Protege pour montrer le fonctionnement des fonctions built-in
#permettant l'accès aux attributs d'un objet
class Protege:

    #constructor
    def __init__(self):
        self.a = 1
        self.b = 2
        self.c = 3

    #methods
    ##magic methods
```

```
        def __getattr__(self, attribut):
            print("Alerte ! Il n'y a pas d'attribut {} ici !".format(attribut))

        def __setattr__(self, attribut, valeur):
            object.__setattr__(self, attribut, valeur)

        def __delattr__(self, attribut):
            raise AttributeError("Vous ne pouvez supprimer aucun attribut " \
            + "de cette classe !")

    pro = Protege()
    print(getattr(pro, "a")) #1

    setattr(pro, "a", 12)
    print(getattr(pro, "a")) #12

    print(hasattr(pro, "b")) #True
    print(hasattr(pro, "e")) #Alerte ! Il n'y a pas d'attribut e ici !\nFalse

    delattr(pro, "c")
    #AttributeError exception levée avec le message
    #"Vous ne pouvez supprimer aucun attribut de cette classe !"
    ######################################################################

    #classe ZDict spécialisant les méthodes spéciales des conteneurs
    class ZDict:

        #constructor
        def __init__(self):
            self._dictionnaire = {}

        #methods
        ##magic methods
        def __getitem__(self, index):
            return self._dictionnaire[index]

        def __setitem__(self, index, valeur):
            self._dictionnaire[index] = valeur

        def __delitem__(self, index):
            print("Vous avez supprimé la paire \"{}\": {}".format(index, \
            self._dictionnaire[index]))
            del self._dictionnaire[index]

        def __contains__(self, objet):
            return objet in self._dictionnaire

        def __len__(self):
            return len(self._dictionnaire)

        def __str__(self):
            return str(self._dictionnaire)

    zdict = ZDict()
    print(zdict) #{}
    print("taille du dictionnaire = {}".format(len(zdict))) #taille du dictionnaire = 0

    zdict["dude"] = 12
    zdict["dudette"] = 24

    print(zdict) #{"dude": 12, "dudette": 24}
    print("dude est une clé ? {}".format("dude" in zdict)) #dude est une clé ? True
    print("taille du dictionnaire = {}".format(len(zdict))) #taille du dictionnaire = 2

    print("zdict[\"dude\"] = {}".format(zdict["dude"])) #zdict["dude"] = 12

    del zdict["dude"] #Vous avez supprimé la paire "dude": 12
```

```
print(zdict) #{"dudette": 24}
print("dude est une clé ? {}".format("dude" in zdict)) #dude est une clé ? False
print("taille du dictionnaire = {}".format(len(zdict))) #taille du dictionnaire = 1
################################################

#classe Duree spécialisant les méthodes spéciales
#des opérateurs arithmétiques et de comparaison
class Duree:

    #constructor
    def __init__(self, min=0, sec=0):
        self.min = min
        self.sec = sec

    #methods
    ##instance methods
    def nb_secondes_total(self):
        return self.sec + self.min*60

    ##magic methods
    def __str__(self):
        return "{0:02}:{1:02}".format(self.min, self.sec)

    #addition
    def __add__(self, objet_a_ajouter):
        """objet_a_ajouter est un entier"""

        nouvelle_duree = Duree()
        nouvelle_duree.min = self.min
        nouvelle_duree.sec = self.sec + objet_a_ajouter

        if nouvelle_duree.sec >= 60:
            nouvelle_duree.min += nouvelle_duree.sec // 60
            nouvelle_duree.sec %= 60

        return nouvelle_duree

    #addition (right associativity)
    def __radd__(self, objet_a_ajouter):
        """objet_a_ajouter est un entier"""
        return self + objet_a_ajouter
        #l'opérateur + est commutative donc
        #self + objet_a_ajouter = objet_a_ajouter + self

    #shortcut addition
    def __iadd__(self, objet_a_ajouter):
        """objet_a_ajouter est un entier"""
        self.sec += objet_a_ajouter

        if self.sec >= 60:
            self.min += self.sec // 60
            self.sec %= 60

        return self

    #subtraction
    def __sub__(self, objet_a_retirer):
        """objet_a_retirer est un entier"""

        nouvelle_duree = Duree()
        nouvelle_duree.min = self.min
        nouvelle_duree.sec = self.sec - objet_a_retirer

        if nouvelle_duree.sec < 0:
            nouvelle_duree.sec = 60 - (objet_a_retirer%60)
            nouvelle_duree.min -= (objet_a_retirer//60 + 1)
```

```
            return nouvelle_duree

        #logical equality
        def __eq__(self, autre_duree):
            return self.sec == autre_duree.sec and self.min == autre_duree.min

        #greather than
        def __gt__(self, autre_duree):
            secondes_self = self.nb_secondes_total()
            secondes_autre_duree = autre_duree.nb_secondes_total()
            return secondes_self > secondes_autre_duree

d1 = Duree(12, 8)
print(d1) #12:08

d2 = d1 + 54 #d1 + 54 secondes
print(d2) #13:02

d3 = 52 + d1
print(d3) #13:00

d4 = d3 - 121
print(d4) #10:59

d4 += 128
print(d4) #13:07

d5 = Duree(12, 8)
print("d1 == d5 ? {}".format(d1 == d5)) #d1 == d5 ? True
print("d3 == d4 ? {}".format(d3 == d4)) #d1 == d4 ? False
print("d2 > d1 ? {}".format(d2 > d1)) #d2 > d1 ? True
print("d3 > d4 ? {}".format(d3 > d4)) #d3 > d4 ? False
############################################################

#classe Temp spécialisant les méthodes spéciales
#__getstate__() et __setstate__() du module pickle
from pickle import Pickler, Unpickler

class Temp:
    #constructor
    def __init__(self):
        self.a1 = "une valeur"
        self.a2 = "une autre valeur"
        self.temp = 5

    #methods
    ##magic methods
    def __getstate__(self):
        #on crée une copie du dictionnaire des attributs de l'objet courant
        #pour ne pas affecter la valeur de l'attribut temp de l'objet
        dict_attr = dict(self.__dict__)
        dict_attr["temp"] = 0 #annuler la valeur de temp avant la sérialisation

        return dict_attr #l'objet sérialisé par pickle

    def __setstate__(self, dict_attr):
        dict_attr["temp"] = 10
        self.__dict__ = dict_attr

    def __str__(self):
        return "{}, {}, {}".format(self.a1, self.a2, self.temp)

temp = Temp()
print("temp avant sérialisation : {}".format(temp)) #temp avant sérialisation : 5

with open("data", "wb") as donnees_ecriture:
    myPickler = Pickler(donnees_ecriture)
```

```
    myPickler.dump(temp)

with open("data", "rb") as donnees_lecture:
    myUnpickler = Unpickler(donnees_lecture)
    print("temp après sérialisation : {}".format(myUnpickler.load())) #temp après sérialisation : 10
```

Le tri en Python

Rappel sur les tris

- `l.sort() -> None` : renvoie la **liste** `l` **triée par ordre croissant** selon l'**ordre naturel** sur ses **éléments**
- `sorted(sequence[, ...])` : fonction renvoyant une **copie** de la **sequence** `sequence` **triée** par **ordre croissant** (*par défaut*) selon l'**ordre naturel** défini sur ses **éléments**
- l'**ordre naturel** sur les **éléments** d'une **séquence** est défini par **Python** via l'**opérateur** `<` à travers la **méthode spéciale** `__lt__()`
- l'**ordre naturel** d'une **séquence** dépend du **type** de ses **éléments**
- si l'on essaie de trier une **séquence** contenant des **éléments** de **types différents**, l'interprète de **Python** ne saura pas le faire et levera une **exception** de type `TypeError` ayant comme **message** " `<` *not supported between instances of ...*"

Utilisation avancée des tris

Trier selon l'attribut `key`

1. `key` : **attribut nommé optionnel** passé à la **fonction built-in** `sorted()` (ou la **méthode des listes** `sort()`) et prenant comme **valeur** une **fonction de tri** (une **fonction lambda** ou une **déclaration de fonction**) définissant l'**ordre de tri sur les éléments**
2. **utilisation** :
   - `l.sort(key=<fonction>[, ...])` : **trier** la **liste** `l` selon la **fonction de tri** `fonction` passée en **argument** de l'**attribut nommé optionnel** `key`
   - `sorted(sequence, key=<fonction>[, ...])` : renvoyer une **copie** de la **séquence** `sequence` **triée** selon la **fonction de tri** `fonction` passée en **argument** de l'**attribut nommé optionnel** `key`

Trier selon l'attribut `reverse`

1. `reverse` :
   - **attribut nommé optionnel** passé à la **fonction built-in** `sorted()` (ou la **méthode des listes** `sort()`) pour **trier** les **éléments** par **ordre décroissant** selon l'**ordre défini** sur les **éléments**
   - **valeur par défaut** : `False`
2. **utilisation** :

- `l.sort(reverse=True[, ...])` : **trier** la **liste** `l` par **ordre décroissant** selon l'**ordre naturel** défini sur ses **éléments**
- `sorted(sequence, reverse=True[, ...])` : renvoyer une **copie** de la **séquence** `sequence` **triée** par **ordre décroissant** selon l'**ordre naturel** défini sur ses **éléments**

remarque on peut **combiner l'utilisation des attributs nommés optionnels** pour **changer les critères du tri**

Trier des objets personnalisés

on peut **trier** les **séquences de notre classe** en :

1. **définissant l'ordre naturel des éléments** en **surchargeant l'opérateur** `<` via la définition de la **méthode spéciale** `__lt__()`

2. **passant** une **fonction d'ordre** à l'**argument nommé optionnel** `key` de la **fonction** `sorted()`

Le module `operator`

1. non importé automatiquement par l'interprète : `import operator`
2. **méthodes** :
   - `operator.itemgetter(item) -> itemgetter object` : **trier** la **séquence** selon son **élément** `item`
   - `operator.attrgetter(attr[, ...]) -> attrgetter object` : **trier** la **séquence** selon l'**attribut**/les **attributs** `attr[, ...]` de l'**objet élément** de la **séquence**

Remarque

Le **tri** en **Python** est **stable** (*l'ordre de deux éléments dans la liste ne sera pas modifié s'ils sont égaux*) => possibilité de **chaîner les tris**

Exemples

```
#tri d'une liste de tuples représentant des étudiants
#selon l'ordre naturel et selon un ordre défini par une fonction
etudiants = [
("Clément", 14, 16),
("Charles", 12, 15),
("Oriane", 14, 18),
("Thomas", 11, 12),
("Damien", 12, 15)
]

print("sans tri :", etudiants)
print("tri par ordre naturel sur les noms :", sorted(etudiants))
print("tri par ordre naturel sur leurs moyennes :", sorted(etudiants, key=lambda element: element[2]))
#####################################

#tri par ordre décroissant d'une liste d'objets de type Etudiant
#selon l'ordre défini par une fonction
class Etudiant:

    def __init__(self, prenom, age, moyenne):
        self.prenom = prenom
        self.age = age
        self.moyenne = moyenne

    def __repr__(self):
        return "Etudiant {} (age={}, moyenne={})".format(self.prenom, self.age, self.moyenne)

etudiants = [
Etudiant("Clément", 14, 16),
Etudiant("Charles", 12, 15),
Etudiant("Oriane", 14, 18),
Etudiant("Thomas", 11, 12),
Etudiant("Damien", 12, 15)
]

print("sans tri :", etudiants)
print("tri par ordre décroissant selon les moyennes :", sorted(etudiants, key=lambda etudiant: etudiant.moy
#####################################

#tri d'une liste d'objets de type Etudiant et d'une liste de tuples
#désignant des étudiants en utilisant les méthodes du module operator
from operator import itemgetter, attrgetter

class Etudiant:
```

```
        #constructor
        def __init__(self, prenom, age, moyenne):
            self.prenom = prenom
            self.age = age
            self.moyenne = moyenne

        #methods
        ##magic methods
        def __repr__(self):
            return "Etudiant {} (age={}, moyenne={})".format(self.prenom, \
            self.age, self.moyenne)

    etudiants = [
    ("Clément", 14, 16),
    ("Charles", 12, 15),
    ("Oriane", 14, 18),
    ("Thomas", 11, 12),
    ("Damien", 12, 15)
    ]

    etudiants_objets = [
    Etudiant("Clément", 14, 16),
    Etudiant("Charles", 12, 15),
    Etudiant("Oriane", 14, 18),
    Etudiant("Thomas", 11, 12),
    Etudiant("Damien", 12, 15)
    ]

    print("tri des tuples selon leurs moyennes:", sorted(etudiants_tuples, key=itemgetter(2)))
    print("tri des objets selon leurs ages et puis leurs moyennes :", sorted(etudiants_objets, key=attrgetter("
    ################################################

    #tri d'une liste d'objets de type LigneInventaire selon plusieurs critères
    #en utilisant la méthode attrgetter() du module operator
    from operator import attrgetter
    class LigneInventaire:
        """Classe représentant une ligne d'un inventaire de vente.

        Une ligne d'inventaire est caractérisée par :
        1. le nom du produit
        2. le prix unitaire du produit
        3. la quantité vendue du produit"""

        #constructor
        def __init__(self, produit, prix, quantite):
            self.produit = produit
            self.prix = prix
            self.quantite = quantite

        #methods
        ##magic methods
        def __repr__(self):
            return "<Ligne d'inventaire {} ({}x{})".format(self.produit, \
            self.prix, self.quantite)

    inventaire = [
    LigneInventaire("pomme rouge", 1.2, 19),
    LigneInventaire("orange", 1.4, 24),
    LigneInventaire("banane", 0.9, 21),
    LigneInventaire("poire", 1.2, 24)
    ]

    print("tri par prix puis par quantité :", sorted(inventaire, key=attrgetter("prix", "quantite")))

    inventaire_trie = sorted(inventaire, key=attrgetter("quantite"), reverse=True)
    print("tri par quantite (ordre décroissant) puis par prix (ordre croissant)", sorted(inventaire_trie, key=a
```

L'héritage

Introduction

1. **relation d'héritage** entre **classes** :
   - **classe** `A` **hérite** d'une **classe** `B` si `A` **est** un `B` mais `B` **n'est pas forcément** un `A`
   - on dit que `A` est la **sous-classe** de `B` et `B` **est** la **superclasse** de `A`
2. en **Python** :
   - **héritage simple** et **multiple**
   - **toutes les classes héritent** de la **classe** `object` qui définit **toutes les méthodes spéciales**

Héritage simple

1. une **classe hérite** d'**une seule superclasse directe**
2. lors de l'**appel d'une méthode** sur un **objet**, l'interprète **Python recherche** cette **méthode** depuis la **classe** de **l'objet** :
   - *s'il la trouve* **c'est bon**,
   - *sinon* il commence à **rechercher la méthode récursivement** dans les **superclasses de la classe de l'objet en commençant depuis sa superclasse directe**
3. lors de l'**appel d'un constructeur** sur un **objet** :
   - *si* le **constructeur de la classe** de **l'objet** est **défini**, on l'appelle
   - *sinon* on appelle le **constructeur de sa superclasse la plus proche dans l'hiérarchie d'héritage**
   - *sinon* on appelle le **constructeur de la classe** `object`
4. on peut appeler une **implémentation d'une méthode héritée d'une superclasse** et **redéfinie** dans la **sous-classe** en utilisant la notation : `SuperClasse.methode([...])`

Syntaxe

```
class ClasseFille(ClasseMere):
  #définition de la classe

  #constructor
  def __init__(self[, ...]):
    #il faut appeler le constructeur de la superclasse explicitement
    #pour initialiser les attributs hérités
    ClasseMere.__init__([...])
```

Héritage multiple

1. une **classe hérite de plusieurs superclasses directes**
2. lors de l'appel d'une **méthode sur un objet** : *idem* que l'**héritage simple**, mais **en commençant** depuis la **première superclasse directe déclarée**, puis la **deuxième** si on ne l'a trouve pas dans **l'hiérarchie d'héritage de la première superclasse directe**, puis la **troisième** ...
3. lors de l'appel d'un **constructeur sur un objet** : *idem* que **l'héritage simple**

Syntaxe

```
class ClasseFille(ClasseMere1, ClasseMere2[, ...]):
  #définition de la classe
```

Exemples

```
#Exemple d'héritage simple
class Personne:

    def __init__(self, nom):
        self.nom = nom
        self.prenom = "Martin"

    def __str__(self):
        return "a = {}; b = {}".format(self.prenom, self.nom)

class AgentSpecial(Personne):

    def __init__(self, nom, matricule):
        Personne.__init__(self, nom)
        self.matricule = matricule

    def __str__(self):
          return "Agent {} {}, matricule {}".format(self.prenom, self.nom, self.matricule)

agent = AgentSpecial("Fisher", "18327-121")
print(agent) #Agent Martin Fisher, matricule 18327-121

print("AgentSpecial sous-classe de Personne :", issubclass(AgentSpecial, Personne)) #True
print("Personne sous-classe de AgentSpecial :", issubclass(Personne, AgentSpecial)) #False
print("AgentSpecial sous-classe de object :", issubclass(AgentSpecial, object)) #True
print("Personne sous-classe de Personne :", issubclass(Personne, Personne)) #True

print("agent instance de AgentSpecial :", isinstance(agent, AgentSpecial)) #True
print("agent instance de Personne :", isinstance(agent, Personne)) #True
print("agent instance de object :", isinstance(agent, object)) #True
```

Les itérateurs et les générateurs

Les itérateurs

1. **définition** : un **itérateur** est un **objet** se chargeant de **parcourir les éléments d'un objet** dit **itérable** (*conteneur, séquence, ...*)
2. **méchanisme de la boucle** `for` :
   - quand **l'interprète Python** tombe sur une **boucle** `for` il **appelle l'itérateur de l'objet itérable courant** qu'on souhaite parcourir.
   - l'**itérateur** permettant de **parcourir un objet itérable** est **retourné** par la **méthode spéciale** `__iter__()` définie dans la classe de l'objet itérable
   - à chaque **tour** de la **boucle** `for` , l'interprète appelle la **méthode** `__next__()` de l'**itérateur** qui :
     a. *soit* **renvoie l'élément suivant du parcours**
     b. *soit* **lève l'exception** `StopIteration` si le **parcours touche sa fin**
3. `iter(objet)` : une **fonction built-in** permettant d'appeler la **méthode spéciale** `__iter__()` de la **classe de l'objet** `objet`
4. `next(iterator)` : une **fonction built-in** permettant d'appeler la **méthode spécale** `__next__()` de l'**itérateur** `iterator`

Les générateurs

1. **problème** : lorsqu'on **définit** des **objets itérables**, parfois on aura besoin de leur **parcourir de plusieurs manières différentes** -> leur **définir un itérateur pour chaque parcours** -> **redondance de code** (*i.e. une classe pour chaque itérateur spécialisant chacune sa propre méthode* `__next__()`)
2. **solution** : utiliser des **générateurs** : un moyen **plus simple** et **moins verbeux** pour **définir des itérateurs**

Les générateurs simples

1. **définition** : un **générateur simple** est :
   - un **objet itérateur** retourné par une **fonction** définissant un **type de parcours spécifique** de l'**objet itérable** (*et donc sa propré méthode* `__iter__()`)
   - la **définition des itérations** du **générateur** retourné utilise le **mot-clé** `yield` dans le corps de la **fonction le définissant** afin de **renvoyer** une **valeur marquant la fin de l'itération** (*mot-clé utilisable uniquement dans une fonction*)
2. **méchanisme** :
   - la **fonction** commence son **exécution** quand on demande le **premier élément du parcours** en invoquant la **fonction** `next()` sur le **générateur** retourné par la **fonction** (*qui appellera la méthode* `__iter__()` *du générateur*)
   - quand elle trouve le **mot-clé** `yield` , elle **renvoie** la **valeur** qui suit et se met en **pause**
   - quand on demande le **prochain élément** via `next()` , la **fonction continue son exécution** dans la **prochaine itération depuis l'instruction suivant l'instruction contenant** `yield` **qui a mis son exécution en pause**
   - ...
   - à la **fin de l'exécution de la fonction**, l'**exception** `StopIteration` est **automatiquement levée** par l'interprète **Python**

Les générateurs comme co-routines

1. **co-routine** : un moyen permettant d'**altérer dynamiquement** un **objet itérable** *pendant* son parcours
2. le **système des co-routines** est contenu dans le **mot-clé** `yield` et l'utilisation de **certaines méthodes de générateurs**
3. **méthodes de co-routines** :

- `<generator>.close()` : **interrompre le parcours d'un objet itérable** par un **générateur** `generator`
- `<generator>.send(valeur)` : **envoyer** une **donnée** `valeur` à un **générateur** (*pendant son exécution*) pour **affecter le parcours de l'objet itérable**

Exemples

```
#Exemple de base sur les itérateurs
chaine = "test"
iterateur_chaine = iter(chaine)
print(iterateur_chaine) #<str_iterator object at 0x7fadd007e128>

print(next(iterateur_chaine)) #t
print(next(iterateur_chaine)) #e
print(next(iterateur_chaine)) #s
print(next(iterateur_chaine)) #t
print(next(iterateur_chaine)) #StopIteration exception levée
####################################

#création d'itérateur permettant de parcourir un string en inverse
#dans la boucle for en utilisant une classe wrapper pour la class str
class IterRevStr:
    """Classe définissant un itérateur
    permettant de parcourir un string en inverse"""

    #constructor
    def __init__(self, chaine):
        self.chaine = chaine
        self.position = len(chaine)

    #methods
    ##NEXT ELEMENT METHOD
```

```
        def __next__(self):
            if self.position == 0:
                raise StopIteration
            self.position -= 1
            return self.chaine[self.position]

    class RevStr(str):
        """Classe reprenant les méthodes et attributs des strings
        à la différence de la méthode de parcours qui sera redéfinie
        pour parcourir le string en inverse (de sa fin à son début)"""

        #methods
        ##ITERATION METHOD REDEFINITION
        def __iter__(self):
            return IterRevStr(self)

    chaine = RevStr("Bonjour")
    print(chaine) #Bonjour
    for lettre in chaine:
        print(lettre) #r\nu\no\nj\nn\no\nB
    ####################################

    #Exemple de base sur les générateurs en utilisant explicitement les méthodes
    #iter() et next() pour faire appel à l'itérateur généré par le générateur
    def generator_factory():
        """Notre premier générateur. Il va simplement renvoyer 1, 2, et 3"""

        yield 1
        yield 2
        yield 3

    print(generator_factory) #<function generator_factory at 0x7f0d5abe1ea0>
    print(generator_factory()) #<generator object generator_factory at 0x7f0d5ab03410>

    generator = generator_factory()
    print(next(generator)) #1
    print(next(generator)) #2
    print(next(generator)) #3
    print(next(generator)) #StopIteration levée

    #Ou par une boucle for
    for nombre in generator_factory():
        print(nombre) #1\n2\n3
    ####################################

    #générateur permettant de parcourir un string de gauche à droite
    def string_generator_factory(string):
        for letter in string:
            yield letter

    string = "test"
    for letter in string_generator_factory(string):
        print(letter) #t\ne\ns\nt
    ########################################

    #générateur permettant de parcourir un string de droite à gauche (sens inverse)
    def inverse_string_generator_factory(string):
        letter_index = len(string) - 1
        while letter_index >= 0:
            yield string[letter_index]
            letter_index -= 1

    string = "Bonjour"
    for letter in inverse_string_generator_factory(string):
        print(letter) #r\nu\no\nj\nn\no\nB

    #création d'un générateur permettant de parcourir un intervalle
```

```
#(bornes non incluses) d'un objet itérable
def intervalle(inf, sup):
    """Générateur parcourant la série des entiers entre inf et sup"""

    if inf < sup:
        inf += 1
        while inf < sup:
            yield inf
            inf += 1
    elif inf == sup:
          yield inf
          inf += 1
    else:
        inf -= 1
        while inf > sup:
            yield inf
            inf -= 1

print("Si inf < sup :")
for nombre in intervalle(5, 10):
    print(nombre) #6\n7\n8\n9

print("Si inf > sup :")
for nombre in intervalle(10, 5):
    print(nombre) #9\n8\n7\n6

print("Si inf == sup :")
for nombre in intervalle(10, 10):
    print(nombre) #10
#####################################

#création d'un générateur qui sera interrompu pendant son parcours
#par une méthode représentant un co-routine (close())
def intervalle(inf, sup):
    """Générateur parcourant la série des entiers entre inf et sup (non inclus)
    Le générateur doit pouvoir sauter une certaine plage de nombres
    en fonction d'une valeur qu'on lui donne pendant le parcours.
    La valeur qu'on lui donne sera la valeur de inf

    inf doit être < sup"""

    inf += 1
    while inf < sup:
        yield inf
        inf += 1

generateur_intervalle = intervalle(5, 20)
for nombre in generateur_intervalle:
    if nombre >= 17:
        generateur_intervalle.close() #interruption du générateur
    print(nombre, end=" ") #6 7 8 9 10 11 12 13 14 15 16 17 18
#####################################

#création d'un générateur permettant de parcourir un intervalle
#(bornes non incluses) d'un objet itérable mais en le modifiant
#pendant le parcours via un co-routine
#(à travers "yield" et la méthode de générateur send())
def intervalle(inf, sup):
    """Générateur parcourant la série des entiers entre inf et sup (non inclus)
    Le générateur doit pouvoir sauter une certaine plage de nombres
    en fonction d'une valeur qu'on lui donne pendant le parcours.
    La valeur qu'on lui donne sera la valeur de inf

    inf doit être < sup"""

    inf += 1
    while inf < sup:
```

```
        valeur_recue = (yield inf)
        if valeur_recue is not None:
            inf = valeur_recue
        inf += 1

generateur_intervalle = intervalle(10, 25)
for nombre in generateur_intervalle:
    if nombre == 15:
        generateur_intervalle.send(20)
    print(nombre, end=" ") #11 12 13 14 15 22 23 24
```

TP : Un dictionnaire ordonné

```
#!/usr/bin/python3
from operator import itemgetter
class DictionnaireOrdonne:

    #constructor
    def __init__(self, dictionnaire={}, **couples):
        self._k = []
        self._v = []

        if dictionnaire is not {}:
            if type(dictionnaire) in (dict, DictionnaireOrdonne):
                for key, value in dictionnaire.items():
                    self._k.append(key)
                    self._v.append(value)
            else:
                raise TypeError("Type de paramètre attendu = dict, " \
                + "alors que type reçu = {}".format(type(dictionnaire)))

        for key, value in couples.items():
            self._k.append(key)
            self._v.append(value)

    #methods
    ##instance methods
    def keys(self):
        """Liste des clés du dictionnaire ordonné"""
        return list(self._k)

    def values(self):
        """Liste des valeurs du dictionnaire ordonné"""
        return list(self._v)

    def items(self):
        """Liste des items (clé, valeur) du dictionnaire ordonné"""
        for i, key in enumerate(self._k):
            value = self._v[i]
            yield (key, value)

    def sort(self, reverse=False):
        """Trier le dictionnaire ordonné par ordre croissant
        selon l'ordre naturel sur ses clés"""
        self_sorted_items = sorted(self.items(), key=itemgetter(0), \
        reverse=reverse)
        index = 0

        for key, value in self_sorted_items:
            self._k[index] = key
            self._v[index] = value
            index += 1
```

```
        def reverse(self):
            """Trier le dictionnaire ordonné par ordre décroissant
            selon l'ordre naturel sur ses clés"""
            self.sort(reverse=True)


        ##magic methods
        def __repr__(self):
            """Méthode spéciale pour afficher le dictionnaire ordonné"""
            dictionnaire_ordonne = dict()
            for i, key in enumerate(self._k):
                dictionnaire_ordonne[key] = self._v[i]
            return repr(dictionnaire_ordonne)

        def __str__(self):
            """Méthode spéciale pour afficher le dictionnaire ordonné
            ou le convertir en string"""
            return repr(self)

        def __contains__(self, key):
            """Méthode spéciale pour tester si "key" est une clé
            du dictionnaire ordonné"""
            return key in self._k

        def __getitem__(self, key):
            """"Méthode spéciale pour surcharger l'opérateur [] en lecture
            (renvoyer la valeur dictionnaire[key])"""
            if key not in self._k:
                raise KeyError("La clé {} n'est pas " \
                + "dans le dictionnaire ordonné !".format(key))
            index_key = self._k.index(key)
            return self._v[index_key]

        def __setitem__(self, key, value):
            """Méthode spéciale pour surcharger l'opérateur [] en écriture
            (dictionnaire[key] = value)

            Si la clé "key" est déjà dans le dictionnaire,
            la valeur associée sera écrasée par "value"
            Sinon, la paire "key", "value" sera ajoutée en fin du
            dictionnaire ordonné"""
            if key in self._k:
                index_key = self._k.index(key)
                self._v[index_key] = value
            else:
                self._k.append(key)
                self._v.append(value)

        def __delitem__(self, key):
            """Méthode spéciale pour surcharger l'opérateur [] en suppression
            (del dictionnaire[key])
            qui supprimera la clé "key" et sa valeur associée
            dans le dictionnaire ordonné"""
            if key in self._k:
                index_key = self._k.index(key)
                del self._k[index_key]
                del self._v[index_key]
            else:
                raise KeyError("La clé {} n'est pas " \
                + "dans le dictionnaire ordonné !".format(key))

        def __len__(self):
            """Méthode spéciale pour renvoyer la taille du dictionnaire ordonné"""
            return len(self._k)

        def __iter__(self):
            """Méthode spéciale pour définir un itérateur
```

```
            sur le dictionnaire ordonné afin de la parcourir via une boucle for"""
            return iter(self._k)

        def __add__(self, dico_ajoute):
            """Méthode spéciale permettant de surcharger l'opérateur +
            pour les dictionnaires ordonnés

            Cette opération consiste à ajouter les paires clés/valeur
            du dictionnaire ordonné "dico_ajoute"
            à celles du dictionnaire ordonné courant"""
            if type(dico_ajoute) is not type(self):
                raise TypeError("Impossible de concaténer {} et " \
                + "{}".format(type(self), type(dico_ajoute)))
            else:
                nouveau = DictionnaireOrdonne(self)
                for key in dico_ajoute:
                    nouveau[key] = dico_ajoute[key]

            return nouveau

        def __iadd__(self, dico_ajoute):
            """Méthode spéciale permettant de surcharger l'opérateur +=
            pour les dictionnaires ordonnés"""
            nouveau = self + dico_ajoute
            return nouveau

    #TEST
    #####
    fruits = DictionnaireOrdonne()
    print(fruits)

    fruits["pomme"] = 52
    fruits["poire"] = 34
    fruits["prune"] = 128
    fruits["melon"] = 15

    print(fruits)

    fruits.sort()
    print(fruits)

    legumes = DictionnaireOrdonne(carotte=26, haricot=48)
    print(legumes)
    print(len(legumes))
    legumes.reverse()
    print(legumes)

    fruits += legumes
    print(fruits)

    del fruits["haricot"]
    print("haricot" in fruits)
    print(legumes["haricot"])

    for key in legumes:
        print(key)

    print(legumes.keys())
    print(legumes.values())

    for nom, qtt in legumes.items():
        print("{0} ({1})".format(nom, qtt))
```

Les décorateurs

Introduction

1. **définition** :
   - un **concept de la métaprogrammation en Python**
   - des **fonctions** pour **modifier le comportement par défaut d'autres fonctions/classes** (prenant en **paramètre** la **fonction/classe** et **renvoyant une fonction/classe**)
2. **méchanisme** :
   - une **fonction/classe modifiée par un/plusieurs décorateur(s)** appellera **le(s) décorateur(s)** lors de **son définition**
   - le(s) **décorateur(s) décidera/décideront dans quelles conditions il(s) exécutera/exécuteront** ou pas la/le **fonction/constructeur de la classe**

Décorateurs de fonctions

Décorateurs simples

1. **syntaxe d'utilisation** :

```
@decorateur1
@decorateur2

@decorateurN
def fonction():
  #code

# \equiv
fonction = decorateur1(decorateur2(...(decorateurN(fonction))))
```

remarques :

- le **décorateur** prend en **paramètre** une **fonction** qu'il **modifiera** (*ou pas*) et **renvoie** une **fonction** (*qui peut être identique à celle passée en argument ou différente d'elle*)
- si la **fonction renvoyée** par un **décorateur** n'est pas **la même** que celle **décorée** par lui, alors la **fonction retournée** peut être **définie** dans le **corps** du **décorateur** (*vu qu'elle sera utilisée uniquement par lui*)

Décorateurs avec paramètres

syntaxe d'utilisation :

```
@decorateur1(arguments1)
@decorateur2(arguments2)
...
@decorateurN(argumentsN)
  def fonction([*parametres_non_nommes, **parametres_nommes]):
    #code

  #\equiv

  fonction = decorateur1(arguments1)(decorateur2(arguments2) \
  (...(decorateurN(argumentsN)(fonction))))

  #le décorateur "decorateur_i" défini aura comme arguments "arguments_i" et
  #renvoie un décorateur
  #le dernier décorateur sera exécuté sur la fonction "fonction" et renverra
  #une fonction à l'avant dernier décorateur et ainsi de suite
  #si la fonction contrôlée par les décorateurs prend des paramètres
  #(nommés ou non), on doit les passer aux fonctions renvoyées
  #par les décorateurs
```

```
    #(fonction_modifiee_i) (*parametres_non_nommes, **parametres_nommes)
```

Décorateurs de classes

1. *idem* que les **décorateurs** de **fonctions**, sauf que l'on passe une **classe** en **paramètre** et nous **retournons** une **autre classe** (*qui peut être elle-même*)
2. les **arguments passés** à une **fonction décorée** par un **décorateur** sont **passés** à la **fonction retournée** par le **décorateur**
3. les **arguments passés** à un **constructeur d'une classe décorée par un décorateur** sont **passés** à la **classe retournée par le décorateur**

Exemples

```
#Exemple de base sur les décorateurs
def mon_decorateur(fonction):
    print("Notre décorateur est appelé avec la fonction " \
    + "{} passée en argument".format(fonction))
    return fonction

@mon_decorateur
def salut():
    print("Salut !")

#Notre décorateur est appelé avec la fonction
#<function salut at 0x7f5ae9ff7840> passée en argument
################################################

#décorateur renvoyant une fonction de substitution
#à celle qui lui est passée en argument
def mon_decorateur(fonction):
    """Décorateur renvoyant une fonction de substitution à la fonction
    "fonction"

    la fonction de substitution affiche un message d'avertissement
    indiquant que l'on a appelé "fonction" et puis on exécute
    cette fonction"""

    def fonction_modifiee():
        print("Attention ! On appelle {}".format(fonction))
        return fonction()

    return fonction_modifiee

@mon_decorateur
def salut():
    print("Salut !")

#salut = mon_decorateur(salut) \equiv @mon_decorateur\ndef salut():
salut() #Attention ! On appelle <function salut at 0x7fc4a3e43598>\nSalut !
print(salut)
#<function mon_decorateur.<locals>.fonction_modifiee at 0x7fc4a3e43840>
################################################

#décorateur "obsolete" permettant d'arrêter l'exécution d'une fonction
#passée en argument si elle est obsolète
def obsolete(fonction):
    """Décorateur levant une exception pour indiquer que la fonction
    "fonction" est obsolète"""
    def fonction_modifiee():
        raise RuntimeError("La fonction {} est obsolète".format(fonction))
```

```
        return fonction_modifiee

    @obsolete
    def fonction_obsolete():
        print("Je suis obsolète !")

    fonction_obsolete()
    # Traceback (most recent call last):
    #   File "test.py", line 15, in <module>
    #     fonction_obsolete()
    #   File "test.py", line 6, in fonction_modifiee
    #     raise RuntimeError("La fonction {} est obsolète".format(fonction))
    # RuntimeError: La fonction <function fonction_obsolete
    #at 0x7f3ce594f598> est obsolète
    ################################################

    #décorateur prenant en paramètre un nombre de secondes qu'il utilise
    #pour contrôler l'exécution d'une fonction et afficher une alerte si
    #le temps d'exécution de la fonction dépasse le nombre de secondes
    #passé en paramètre
    import time

    def controler_temps(secondes):
        """Contrôle le temps mis par une fonction pour s'exécuter.
        Si le temps d'exécution est supérieur à secondes, on affiche une alerte."""

        def decorateur(fonction):
            """Décorateur qui sera executé lors de la définition
            de la fonction à exécuter"""

            def fonction_modifiee():
                """Fonction renvoyée par le décorateur.
                Elle se charge de calculer
                le temps d'exécution de la fonction"""

                avant = time.time()
                #time() : renvoyer le nombre de secondes passés depuis 01"01"1970
                resultat = fonction()
                apres = time.time()
                execution = apres - avant

                if execution >= secondes:
                    print("La fonction {} a mis {} secondes " \
                    + "pour s'exécuter".format(fonction, execution))

                return resultat
            return fonction_modifiee
        return decorateur

    @controler_temps(4)
    def attendre():
        input("Appuyez sur Entrée...")

    attendre()
    #Appuyez sur Entrée...
    #(en tappant rapidement on sort directement sans aucune alerte)
    #Si on attend un petit moment avant de tapper Entrée, on aura :
    #Appuyez sur Entrée...
    #La fonction <function attendre at 0x7f6ffeed5840>
    #a mis 5.93743634223938 secondes pour s'exécuter
    ################################################

    #même contrôleur de temps défini dans l'exemple précédent
    #mais sur une fonction possédant des paramètres en entrée
    import time

    def controler_temps(secondes):
```

```
        """Contrôle le temps mis par une fonction pour s'exécuter.
        Si le temps d'exécution est supérieur à secondes, on affiche une alerte."""

        def decorateur(fonction):
            """Décorateur qui sera executé lors de la définition
            de la fonction à exécuter"""

            def fonction_modifiee(*parametres_positionnels, **parametres_nommes):
                """Fonction renvoyée par le décorateur.
                Elle se charge de calculer le temps d'exécution de la fonction"""

                avant = time.time()
                #time() : renvoyer le nombre de secondes passés depuis 01"01"1970
                resultat = fonction(*parametres_positionnels, **parametres_nommes)
                apres = time.time()
                execution = apres - avant

                if execution >= secondes:
                    print("La fonction {} a mis {} secondes pour " \
                    + "s'exécuter".format(fonction, execution))

                return resultat
            return fonction_modifiee
        return decorateur

    @controler_temps(4)
    def attendre(nom=None, prenom=None):
        result = ""
        if prenom:
            result += str(prenom)
        if nom:
            result += " " + str(nom).upper()
        input("Appuyez sur entrée {}...".format(result))

    attendre() #Appuyez sur Entrée ...
    attendre("Dude") #Appuyez sur Entrée DUDE...\n
    #La fonction <function attendre at 0x7f4d20818840>
    #a mis 4.404439926147461 secondes pour s'exécuter
    attendre(nom="Dude", prenom="Fuck") #Appuyez sur Entrée Fuck DUDE...
    ################################################

    #décorateur permettant de contrôler les types des paramètres d'une fonction
    def type_controler(*tp_awaited, **tk_awaited):
        """Contrôleur de type des paramètres passés à la fonction contrôlée"""
        def decorator(f):
            """Décorateur qui sera exécuté lors de la définition de la fonction "fonction"
            et renverra la fonction "fonction_modifiee"""
            def f_modif(*p_received, **tk_received):
                """Fonction se chargeant de contrôler les types qu'on lui passe en paramètre
                On commence par vérifier que la taille de la liste des paramètres non nommés attendus est bien
                On contrôle ensuite chaque paramètre reçu.
                    Si le type reçu est le même que celui attendu, tout va bien.
                    Sinon, on lève une exception.
                On répète l'opération sur les paramètres nommés (avec une petite différence, puisqu'il s'agit d

                Si tout va bien (aucune exception n'a été levée),
                on exécute notre fonction en renvoyant son résultat."""

                if len(p_received) != len(tp_awaited):
                    raise TypeError("Error: expected {} arguments, received {} instead".format(len(tp_awaited),

                for i, received in enumerate(p_received):
                    if tp_awaited[i] is not type(received):
                        raise TypeError("Error: received argument \"{}\" at index {} is not of type {}".format(

                for key in tk_received:
                    if key not in tk_awaited:
```

```
                    raise TypeError("Error: received argument \"{}\" has no type defined".format(repr(key))
                if tk_awaited[key] is not type(tk_received[key]):
                    raise TypeError("Error: expected type {} for key {}, received {} instead".format(tk_awa

            return f(*p_received, **tk_received)
        return f_modif
    return decorator

@type_controler(int, int, step=int)
def intervalle(inf, sup, step=1):
    print("Intervalle de {} à {} ayant un écart de {}".format(inf, sup, step))

intervalle(1, 8) #Intervalle de 1 à 8 ayant un écart de 1
intervalle(1, "oups")
# Traceback (most recent call last):
#   File "test.py", line 39, in <module>
#     intervalle(1, "oups")
#   File "test.py", line 22, in f_modif
#     raise TypeError("Error: received argument \"{}\" at index {} is not of type {}".format(received, i, t
# TypeError: Error: received argument "oups" at index 1 is not of type <class 'int'>

intervalle(1, 8, step=2) #Intervalle de 1 à 8 ayant un écart de 2
intervalle(1, 8, step="dude")
# Traceback (most recent call last):
#   File "test.py", line 48, in <module>
#     intervalle(1, 8, step="dude")
#   File "test.py", line 28, in f_modif
#     raise TypeError("Error: expected type {} for key {}, received {} instead".format(tk_awaited[key], key
# TypeError: Error: expected type <class 'int'> for key step, received <class 'str'> instead
################################################

#décorateur simple de classes
def decorateur_classe(classe):
    print("Définition de la classe {}".format(classe))
    return classe

@decorateur_classe
class Test:
    pass
################################################

#décorateur permettant de contrôler les classes singletons
def singleton(classe):
    """Définition d'un décorateur singleton permettant de
    contrôler qu'une classe singleton ne soit pas instanciée
    plus qu'une fois"""
    #dictionnaire associant une classe singleton à sa première instance créée
    instances = {}

    def get_instance():
        if classe not in instances:
            instances[classe] = classe()
            #création de la première instance de la classe singleton "classe"
            #et son association à "classe" dans le dictionnaire
        return instances[classe]
    return get_instance
    #cette fonction sera appelée avant le constructeur de la classe "classe"
    #(pour contrôler la création des instances)

@singleton
class Test:
    pass

a = Test()
b = Test()
print("a is b ? {}".format(a is b)) #a is b ? True
```

Les métaclasses

Introduction

1. **définition** : les **métaclasses** sont des **modèles de classes**. Autrement dit, ce sont des **classes génériques** permettant d'**instancier des classes concrètes**.
2. la **méthode spéciale** `__init__(self[, args...])` :
   - est le **constructeur** d'une **classe** permettant d'**initialiser** l'**instance courante** de la **classe** (`self`) en **initialisant** ses **attributs**.
   - cette **méthode** ne permet pas de **créer l'instance courante**, mais **uniquement de l'initialiser**
3. la **méthode spéciale** `__new__()` :
   - `__new__(cls[, args...])` : **méthode de classe** permettant de **créer l'instance courante** `self` à partir de la **classe** `cls` et de la **passer** [avec les **arguments** `args`] au **constructeur** (`__init__(self[, args...])`) pour l'**initialiser**
   - **méthode définie par la classe** `object` et **héritée** par **toutes les classes**

Créer des classes dynamiquement

1. **rappel** : **tout est objet en Python**
2. **conséquence** : les **classes sont aussi des objets** => les **classes** sont elles-mêmes **modélées** sur des **classes**
3. en **Python**, **toutes les classes sont modélées sur la classe** `type` (`type` est la **métaclasse** de **toutes les classes en Python** (*par défaut*) :
   - quand on crée une **nouvelle classe**, l'interprète **Python** appelle la **méthode** `__new__()` de la **classe** `type`
   - une fois la **classe créee**, on appelle le **constructeur** `__init__()` de la **classe** `type`
4. on peut ainsi utiliser ce fait pour **créer des classes dynamiquement à travers la classe** `type` (*sans utiliser* `class`).
5. le **constructeur de la classe** `type` prend trois **paramètres** :
   - le **nom de la classe** (*string*)
   - un **tuple** contenant les **superclasses** de notre nouvelle **classe**
   - un **dictionnaire** contenant les **attributs** et **méthodes de la classe**
6. **processus de création dynamique de classes avec la classe** `type` :
   - on commence par **créer** les **fonctions** correspondant aux **constructeurs** et autres **méthodes d'instance**
   - on place ces **fonctions** dans un **dictionnaire** telle que **chaque paire** contient une **clé** désignant le **nom de la méthode de la classe** et de **valeur** la **fonction créée y correspondant**
   - on fait appel au **constructeur** de `type` en lui passant les **paramètres y correspondant**
   - **remarque** : les **attributs** mis dans le **dictionnaire envoyé** au **constructeur** de `type` sont des **attributs de classe** et **non d'instance**

La classe `type`

constructeur : `type(nom_classe, (tuple_classes_heritees), {attributs_et_methodes})`

Les métaclasses

1. **définition** : une **métaclasse** est un **modèle générique** permettant d'**instancier** des **classes concrètes**
2. **rappel** : la **métaclasse** `type` est la **métaclasse** de **toutes les classes en Python par défaut**
3. une **classe** peut avoir une **autre métaclasse** que `type`
4. **toutes les métaclasses** qu'on définira **hériteront** de `type`
5. la **métaclasse sera exécutée** lors de l'**instanciation** de la **classe concrète** (*sans construction explicite d'une instance de la classe concrète*)

Création

1. la **création** des **métaclasses** se fait de la **même manière** que les **classes** :
   - on appelle `__new__(metacls, nom, bases, dict)` pour **créer** la **classe instanciée** :
     a. `metacls` : le **nom** de la **métaclasse** servant de **base** de **création** de la **classe concrète**
     b. `nom` : le **nom** de la **classe concrète**
     c. `bases` : **tuple** contenant les **superclasses** de la **classe concrète**
     d. `dict` : **dictionnaire** contenant les **méthodes d'instances** et **attributs de classe** de la **classe concrète**
   - on appelle `__init__(self, nom, bases, dict)` pour **construire** la **classe concrète instanciée**
     a. `self` : la **classe concrète** qu'on vient de **créer** via `__new__()`
     b. `nom` : le **nom** de la **classe concrète**
     c. `bases` : **tuple** contenant les **superclasses** de la **classe concrète**
     d. `dict` : **dictionnaire** contenant les **méthodes d'instances** et **attributs de classe** de la **classe concrète**
2. **syntaxe de création d'une métaclasse** :

```
#création de la métaclasse
class MaMetaClasse(type):
 def __new__(metcls, nom, bases, dict):
   #définition de la méthode
   return type.__new__(metcls, nom, bases, dict)

 def __init__(cls, nom, bases, dict):
   #définition de la méthode
   type.__init__(cls, nom, bases, dict)

 #Définition de la classe concrète instancitée depuis la métaclasse
 #et exécution de la métaclasse
 class MaClasse(metaclass=MaMetaClasse):
   #définition de la classe concrète
```

Exemples

```
#classe Personne redéfinissant la méthode spéciale __new__()
#pour visualiser l'ordre de création et d'initialisation de
#l'instance courante de la classe
class Person:
    def __new__(cls, first, last):
        print("Calling __new__() method of class {}".format(cls))
        return object.__new__(cls)

    def __init__(self, first, last):
        """Constructor of Person working instance
        (attribute initialization)"""
        print("Calling __init__()")
        self.first = first
        self.last = last
        self.age = 23
        self.residency = "Lyon"

    def __repr__(self):
        return "{} {} aged {} years, living in {}".format(self.first, \
        self.last, self.age, self.residency)

person = Person("Doe", "John")
print(person)
# Calling __new__() method of class <class '__main__.Person'>
# Calling __init__()
# John Doe aged 23 years, living in Lyon
```

```
######################################

#création dynamique d'une classe Personne
def initialize_person(person, first, last):
    """Fonction jouant le rôle de constructeur pour
    la classe "Personne" qu'on créera dynamiquement"""
    person.first = first
    person.last = last
    person.age = 21
    person.residency = "Lyon"

def present_person(person):
    """Fonction présentant la personne
    Elle affiche les informations de la personne"""
    return "{} {} aged {}, living in {}".format(person.first, person.last, person.age, person.residency)

methods ={
"__init__": initialize_person,
"__repr__": present_person
}

#création du dictionnaire, contenant les méthodes, qui sera passé au
#constructeur de la classe "type" lors de la création dynamique
#de la classe "Personne"
Personne = type("Personne", (), methods) #création dynamique de la classe "Personne"
john = Personne("John", "Doe")
print(john) #John Doe aged 21, living in Lyon
######################################

#création d'une métaclasse pour la création d'une classe concrète
class MaMetaClasse(type):
    """Exemple d'une métaclasse"""
    def __new__(mcls, nom, bases, dict):
        """Création de la classe concrète"""
        print("On crée la classe {}".format(nom))
        return type.__new__(mcls, nom, bases, dict)

    def __init__(cls, nom, bases, dict):
        """Construction de la classe concrète"""
        print("On construit la classe {}".format(nom))
        type.__init__(cls, nom, bases, dict)

class MaClasse(metaclass=MaMetaClasse):
    pass
######################################

#création d'une métaclasse permettant d'ajouter dynamiquement
#les classes concrètes instanciés de la métaclasse à un dictionnaire
#contenant les paires clé = nom de la classe et valeur = la classe elle-même
trace_classes = {}
#définition du dictionnaire qui contiendra les classes concrètes
#instanciées de la métaclasse définie
class MetaWidget(type):
    """Métaclasse pour les widgets graphiques
    Elle hérite de "type"
    elle écrira dans traces_classes à chaque fois qu'une classe
    sera créée"""
    def __init__(cls, nom, bases, dict):
        type.__init__(cls, nom, bases, dict)
        trace_classes[nom] = cls

class Widget(metaclass=MetaWidget):
    pass

print(trace_classes) #{'Widget': <class '__main__.Widget'}

class Bouton(Widget):
```

```
    pass

print(trace_classes)
#{'Widget': <class '__main__.Widget'>, 'Bouton': <class '__main__.Bouton'>}
```

Annexes

Mots-clé du langage

1. `and` : **et logique**
2. `as` : instruction de **nommage** utilisé pour
   - nommer des **espaces de noms** lors de l'**importation de modules/packages** ;
   - nommer les **messages d'exceptions** lors de leur **capture**
   - nommer le **résultat d'une opération** avec `with`
3. `assert` : instruction utiliser pour **définir des assertions**
4. `break` : instruction de **rupture de l'itération courante d'une boucle** et la **sortie** de cette dernière
5. `class` : instruction de **définition de classes**
6. `continue` : instruction de **rupture de l'itération courante d'une boucle** et passage à l'**itération suivante**
7. `def` : instruction de **définition de fonctions**
8. `del` : **supprimer** des **variables** ou des **éléments** de **conteneurs muables** (*e.g. listes et dictionnaires*)
9. `elif` : instruction d'**enchaînement d'instructions conditionnelles** à partir d'un `if`
10. `else` : instruction spécifiant la **dernière instruction conditionnelle** dans une **chaîne d'instructions conditionnelles**
11. `except` : **clause except** du bloc `try, except, else, finally`
12. `False` : **valeur booléene faux**
13. `finally` : **clause finally** du bloc `try, except, else, finally`
14. `for` : **boucle for**
15. `from` : utilisée **avec l'instruction d'importation d'un module/packager** pour **en importer un élément spécifique** et lui associer à **l'espace de noms courant**
16. `global` :
    - utilisée pour permettre à une **fonction** d'**accéder en lecture/écriture à un objet global**.
    - écrite *au tout début du corps* de la **fonction** suivie du **nom de l'objet global**
17. `if` : **instruction conditionnelle**
18. `import` : **importer un élément d'un module/package**
19. `in` : **tester** si un **élément** est **dans une collection** ou **instancier une variable** depuis une **collection** dans une **boucle for**
20. `is` : **égalité physique** (*égalité de référence*)
21. `lambda` : instruction de **définition de fonctions lambdas** (*expressions fonctionnelles*)
22. `None` : **valeur spéciale** désignant l'**absence de valeur**
23. `nonlocal` :
24. `not` : **non logique**
25. `or` : **ou logique**
26. `pass` : **ne rien faire** (**i.e. instruction vide**)
27. `raise` : **lever une exception** en indiquant le **message à afficher**
28. `return` : **retourner** [un **résultat**] depuis une **fonction**
29. `True` : **valeur booléene vraie**

30. `try` : **clause try** du bloc `try, except, else, finally`
31. `while` : **boucle while**
32. `with` : créer un **gestionnaire de contexte** permettant de **vérifier l'ouverture et la fermeture** des **ressources** lors d'une **E/S** définie dans un **bloc de code**
33. `yield` : instruction utilisée **uniquement dans le corps d'une fonction définissant un générateur** et renvoyant la **valeur qui la suit**