

Python - Deuxième partie

Concepts avancés du langage

Joël Cavat, Adrien Lescourt

2017

joel.cavat@heig-vd.ch, adrien.lescourt@hesge.ch

Thèmes abordés dans ce chapitre :

- Style fonctionnel et compréhension de liste
- Arguments arbitraires et déconstruction de liste
- Le contrôle des types
- Les classes
- Iterateurs et generateurs
- Thread & process
- Futures
- UI (avec tkinter)

FP style et compréhension

Style fonctionnel/déclaratif

fonction déclaratives

```
>>> ma_liste = [1, 2, 3, 4, 5]
>>> map(lambda x: x**2, ma_liste)
[1, 4, 9, 16, 25]
>>> filter(lambda x: x % 2 == 0, ma_liste)
[2, 4]
```

Style fonctionnel/déclaratif

fonction déclaratives

Comment filter une liste transformée ?

Style fonctionnel à l'aide de `map/filter` :

```
>>> ma_liste = [1, 2, 3, 4, 5]
>>> filter(lambda x: x % 2 == 0, \
...         map(lambda x: x**2, ma_liste))
[4, 16]
```

Style impératif :

```
>>> ma_liste = [1, 2, 3, 4, 5]
>>> res = []
>>> for x in ma_liste:
...     if x**2 % 2 == 0:
...         res.append(x**2)
...
```

Style fonctionnel/déclaratif

Style fonctionnel

Mieux : les compréhensions de liste

```
>>> ma_liste = [1, 2, 3, 4, 5]
>>> [x**2 for x in ma_liste if x**2 % 2 == 0]
[4, 16]
```

Les compréhensions

Les compréhensions de liste

Les compréhensions permettent de construire des listes, ensembles ou dictionnaires à partir d'une collection itérable.

La notion de compréhension dérive de la notation ensembliste

Par exemple, la liste des éléments pairs de l'ensemble $\{1, 2, 3, \dots, 100\}$

- Mathématiquement : $\{x \in \{1, 2, 3, \dots, 100\} \mid x \text{ pair}\}$
- En Python : `[x for x in range(1,101) if x % 2 == 0]`

Les compréhensions

Les compréhensions de liste

La liste des combinaisons (x, y) tel que $x \in X$ et $y \in Y$, appelé également produit cartésien $X \times Y$.

Exemple

- $X = \{6, 7, 8, 9, 10, "Va", "Re", "Ro", "As"\}$
- $Y = \{"Pique", "Coeur", "Trefle", "Carreau"\}$
- Résultat : Jeux de cartes traditionnel (Chibre, jass, la bataille, ...)

Modélisation :

- Mathématiquement : $\{(x, y) \in X \times Y\}$
- En Python : `[(x,y) for x in X for y in Y]`
- Résultat :

```
[(6, "Pique"), (6, "Coeur"), ..., ("As", "Carreau")]
```


Les compréhensions

Les compréhensions de liste

L'ensemble des couples $(x, y) \in X^2$ tel que $x + y = 5$ avec
 $X = \{1, 2, \dots, 10\}$

- Mathématiquement : $\{(x, y) \in \{1, 2, 3, \dots, 10\}^2 \mid x + y = 5\}$
- En Python :

```
{(x,y) for x in range(1,11) for y in range(1,11) if x + y == 5}
```
- Résultat :

```
{(1, 4), (2, 3), (3, 2), (4, 1)}
```

Les compréhensions

Les compréhensions de liste

Un dictionnaire des couples (x, y) tel que $x \in \{1, 2, \dots, 10\}$ et $y = x^2$

- L'ensemble en mathématiquement : $\{(x, x^2) \mid x \in \{1, 2, 3, \dots, 10\} \text{ et } x + y = 5\}$
- En Python : `carre = {x : x**2 for x in range(1,11)}`
- Résultat :
`{1 : 1, 2 : 4, 3 : 9, 4 : 16, 5 : 25, ..., 9 : 81, 10 : 100}`

```
>>> carre
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49,
8: 64, 9: 81, 10: 100}
>>> carre[6]
36
```

Les compréhensions

Générateur

Il est possible de générer un Générateur. Permet l'utilisation de collection **paraisseuse**.

```
>>> res = (x**2 for x in range(1000000000))
>>> next(res)
0
>>> next(res)
1
>>> next(res)
4
```

Live coding et présentation :

module itertools collections d'itérateurs inspirés des langages et framework fonctionnels

module collections types de données alternatifs et spécialisés

fonctions intégrées any, all, sorted, ...

Arguments arbitraires

Arguments arbitraires

Les arguments des fonctions peuvent être **arbitraires**. C'est-à-dire qu'il est possible d'écrire une fonction qui prend un nombre d'arguments indéterminé.

```
def test(*args):  
    print(args) # ici args est un tuple
```

Voici quelques exemples d'appels :

```
>>> test()  
()  
>>> test(1,2)  
(1,2)
```

Arguments arbitraires

Voici comment définir la fonction somme :

```
def somme(*args):  
    res = 0  
    for i in args:  
        res += i  
    return res
```

Voici quelques exemple d'appels :

```
>>> somme()  
0  
>>> somme(1,2)  
3  
>>> somme(1,2,3,4)  
10
```

Une liste peut être transformée en une liste d'arguments :

```
>>> somme( *[1,2,3] )  
6
```


Cette notation `*` peut également être utilisée pour **destructurer** une liste :

```
>>> liste = [1,2,3,4,5]
>>> x, *xs = liste
>>> x
1
>>> xs
[2,3,4,5]
>>> liste
[1,2,3,4,5]
```

Arguments arbitraires

Il est également possible de préciser que les arguments sont nommés (keyword argument)

```
def affiche(**kwargs):  
    for k, v in kwargs.items():  
        print(k, ":", v)
```

Voici un exempl d'appel :

```
>>> affiche(id=1, name="Joel", language="python")  
id : 1  
name : Joel  
language : python
```

Un dictionnaire peut être transformé en un ensemble d'arguments nommés :

```
>>> dic = { 'id': 1, 'name': 'Adrien' }  
>>> affiche(**dic)  
id : 1  
name : Adrien
```

Contrôle des types

Python possède un typage **dynamique**. Toutefois, un contrôle des types est réalisé à l'**exécution**.

L'avantage des langages statiques est qu'une grande partie des bugs sont détectés durant la phase de compilation et l'analyse des types.

Mypy est un vérificateur de types qui peut être utilisé pour contrôler nos scripts. Depuis la version 3.6 de Python, il est possible d'utiliser une nouvelle syntaxe pour préciser les types.

Contrôle des types

Quelques exemples de variables typées

```
from typing import List

i: int = 3
nom: str = "Cavat"
xs: List[float] = []
ys: List[int] = [1,2,3]
xs.append(3.4)
```

Les opérations suivantes s'exécutent normalement. Par contre, mypy relève les erreurs :

```
i = nom # ... has type "str", variable has type "int"
xs.append(2) # ok, int <= float
ys.append(2.2) # ... type "float"; expected "int"
```

Le type **Any** permet d'avoir des collections de n'importe quels types :

```
from typing import List, Any  
  
zs: List[Any] = [1,2.2, (3,4), [5,6]]
```

Exemple de fonction typée :

```
def addition(a: float, b: float) -> float:  
    return a + b
```


Le type `Union` permet de préciser qu'une instance peut être de plusieurs types :

```
from typing import Union

def add_conc(a: Union[float, str], \
            b: Union[float, str]) -> Union[float, str]:
    return a + b
```

Contrôle des types

```
from typing import Union

def f(x: Union[int, str]) -> None:
    x + 1          # Error: str + int is not valid
    if isinstance(x, int):
        x + 1      # Here type of x is int.
    else:
        x + 'a'    # Here type of x is str.
```

Il est possible de créer des nouveaux types dérivé sans *overhead* à l'exécution

```
UserId = NewType('UserId', int)
def orders(user_id: UserId) -> List[Order]:
    ...

ls = orders(2) # Failed
ls = orders(UserId(2)) # Ok
```

Programmation orientée objet

Programmation orientée objet

Classes

- Qu'est-ce qu'un Objet ?
- Déclaration d'un objet

```
class MaClass:  
    pass
```

- Instanciation d'un objet

```
mon_object = MaClass()
```

- Tout Python est objet !

Programmation orientée objet

Classes

Une classe est composée d'**attributs** et de **méthodes**.

```
class Fraction:
    def __init__(self, numérateur, dénominateur):
        self.numérateur = numérateur
        self.dénominateur = dénominateur

    def affiche(self):
        print(self.numérateur, '/', self.dénominateur)

    def produit(self, facteur):
        # ... Code du produit ...
```

Programmation orientée objet

Constructeur

Le rôle du **constructeur** est de d'instancier un objet.

C'est à dire d'allouer la mémoire nécessaire à l'objet et d'initialiser ses attributs.

En Python : `__init__(self, param1, param2, etc...)`

```
def __init__(self, numérateur, dénominateur):  
    self.numérateur = numérateur  
    self.dénominateur = dénominateur
```

Programmation orientée objet

Attributs

Les **attributs** sont des variables contenues dans l'objet.

Ils sont toujours précédés par le mot clé **self** qui représente l'objet lui même.

Ils peuvent être déclarés n'importe où dans la classe, mais il est d'usage de **toujours** les déclarer dans le constructeur.

```
def __init__(self, numérateur, dénominateur):  
    self.numérateur = numérateur  
    self.dénominateur = dénominateur
```


Programmation orientée objet

Méthodes

En plus des attributs, une classe contient généralement des **méthodes** d'instances.

Elles doivent impérativement recevoir le paramètre `self` en première position.

```
def affiche(self):  
    print(self.numerateur, '/', self.denominateur)  
  
def produit(self, facteur):  
    # ...Code du produit...
```

Programmation orientée objet

Exemple d'utilisation

```
# Instanciation  
a = Fraction(2, 4)  
b = Fraction(1, 6)  
  
# Utilisation  
a.affiche()   # Accès aux méthodes  
b.numerateur = 4 # Accès aux attributs  
b.affiche()
```

Programmation orientée objet

Méthodes spéciales

En plus des méthodes implémentées par l'utilisateur, une classe contient des méthodes dites **spéciales**.

Comme le constructeur `__init__()`, ces méthodes sont entourées par deux underscores avant et après leur nom.

Ce sont des méthodes qui sont directement appelé par Python dans des contextes précis (affichage, affectation, lecture d'attributs, comparaisons...)

Note :

La fonction `dir()` permet d'afficher tous les attributs et toutes les méthodes d'une classe.

Programmation orientée objet

Représentation

Par exemple pour afficher notre `Fraction` , nous avons créé une méthode `affiche()` . En surchargeant la méthode `__str__()` nous pouvons directement afficher l'objet avec la méthode `print()`

```
def __str__(self):  
    return str(self.numerateur) + ' / '\n  
        + str(self.denominateur)
```

```
a = Fraction(1, 2)  
print(a)
```

Programmation orientée objet

Surcharge d'opérateur

Il est aussi possible d'utiliser directement les **symboles** tels que les opérateurs arithmétiques ou les opérateurs de comparaisons avec notre classe.

Par exemple pour l'addition, le symbole **+** est surchargé avec la méthode `__add__()`

```
a = Fraction(1, 2)
b = Fraction(3, 2)
c = a + b
```

La liste des opérateurs surchargeables est disponible ici :
<https://docs.python.org/3/library/operator.html>

Programmation orientée objet

Type check

Maintenant que nous pouvons surcharger les opérateurs, que ce passerait t'il si l'un des deux membre de l'opération n'est pas un objet `Fraction` ?

Un nombre serait acceptable, mais pas une liste par exemple.

```
a = Fraction(1, 2)
b = 2
c = a * b
```

On verrouille aux types autorisés uniquement

```
def __add__(self, other):
    if not isinstance(other, (int, float, Fraction)):
        raise TypeError
    else:
        # Addition code
```

Programmation orientée objet

Héritage

"En programmation orientée objet, l'héritage est un mécanisme qui permet, lors de la déclaration d'une nouvelle classe, d'y inclure les caractéristiques d'une autre classe." *Wikipedia*

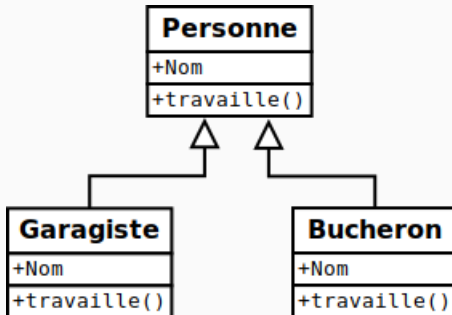
La classe qui hérite d'une autre classe possède les **mêmes attributs** et les **mêmes méthodes** que la classe parent.

```
class Parent:
    def __init__(self):
        self.attribut = 'exemple'
class Enfant(Parent):
    pass

>>> toto = Enfant()
>>> toto.attribut
exemple
```

Programmation orientée objet

Polymorphisme



```
class Personne:
    def __init__(self, nom):
        self.nom = nom

    def travail(self):
        raise NotImplementedError
```


Programmation orientée objet

Polymorphisme

```
class Bucheron(Personne):  
    def travail(self):  
        return self.nom + " coupe les arbres"  
  
class Garagiste(Personne):  
    def travail(self):  
        return self.nom + " répare les voitures"  
  
travailleurs = [Bucheron('Bill'), Garagiste('Carson'),  
                Garagiste('Lee')]  
  
for travailleur in travailleurs:  
    print(travailleur.travail())
```

```
Bill coupe les arbres  
Carson répare les voitures  
Lee répare les voitures
```

Programmation orientée objet

Héritage

On peut remarquer dans l'exemple précédent que les classées héritées n'ont pas de constructeur. Python utilise alors le constructeur de la classe parent.

Si l'on veut leur ajouter un constructeur, l'appel vers le constructeur de la classe parent n'est plus fait automatiquement et doit être explicité. Soit avec le nom de la classe parent, soit avec la fonction `super()`

```
class Garagiste(Personne):  
    def __init__(self, nom, nbr_outils):  
        super().__init__(nom)  
        self.nbr_outils = nbr_outils
```

Programmation orientée objet

Héritage multiple

Python supporte l'héritage multiple, les classes de bases sont alors séparées par des virgules.

```
class C(A, B):  
    def __init__(self):  
        super().__init__()
```

En complément à `isinstance()`, la fonction `issubclass()` permet de savoir si une classe hérite d'une autre.

Programmation orientée objet

Attributs statiques

Les **attributs statiques** sont liées à la classe et non à l'instance.

```
class Test():  
    a = 0
```

Ils sont accessibles depuis la classe ou depuis l'instance.

```
>>> toto = Test()  
>>> Test.a += 1  
>>> Test.a  
1  
>>> toto.a  
1
```

Programmation orientée objet

Méthodes de classes et méthodes statiques

Comme les attributs statiques, les **méthodes de classes** sont liées à la classe elle même.

```
@classmethod
def increment(cls):
    cls.a += 1
```

Elles peuvent être appelées directement depuis la classe ou depuis l'instance

```
>>> toto = Test()
>>> Test.increment()
>>> Test.a
1
>>> toto.increment()
>>> toto.a, Test.a
(2, 2)
```

Live coding **Factory**

Les **properties** permettent le contrôle des accès aux attributs

```
def __init__(self):  
    self._x = None  
  
@property  
def x(self): return self._x  
  
@firstName.setter  
def x(self, val): self._x = val  
  
@firstName.deleter  
def x(self): del self._x
```

Logging en Python

Python fournit une librairie pour le logging.

Documentation :

- formatting :
<https://docs.python.org/3/library/logging.html#logrecord-attributes>
- handlers :
<https://docs.python.org/3/library/logging.handlers.html#module-logging.handlers> (par exemple SysLogHandler pour un log sur serveur distant).

Exemple d'utilisation basique

```
import logging

logging.basicConfig(filename='application.log', \
    level=logging.DEBUG, \
    format='%(asctime)s -- %(name)s \
    -- %(levelname)s -- %(message)s')

logging.debug("coucou")
```

Contenu du fichier 'application.log' :

```
2017-11-05 11:19:56,080 -- root -- DEBUG -- coucou
2017-11-05 11:20:33,556 -- root -- WARNING -- ous
```

Les niveaux :

- CRITICAL : Le programme est dans un état instable
- ERROR : Une opération n'a pas fonctionné
- WARNING : Mérite une attention particulière
- INFO : Informe de la bonne marche du programme
- DEBUG : Debug classique

Decorator

Les **décorateurs** permettent d'ajouter un comportement à du code existant en séparant les préoccupations secondaires de la **logique métier**. (Programmation Orientée Aspect).

Il est possible d'injecter du code avant et après l'appel d'une fonction, permettant ainsi d'**éviter la duplication de code** .

Exemple d'utilisation basique : le logging

```
def log(f):  
    def new_f(*args):  
        logging.debug("avant")  
        for i, arg in enumerate(args):  
            logging.debug(f"Argument {i+1}: {arg}")  
        res = f(*args)  
        logging.info(f"Le résultat est {res}")  
        logging.debug("apres")  
        return res  
    return new_f  
  
@log  
def somme(a, b, c):  
    return a + b + c  
  
print( somme(10,12,23) )
```

Le fichier log après exécution :

```
2017-11-05 12:15:09,218 -- DEBUG -- avant
2017-11-05 12:15:09,218 -- DEBUG -- Argument 1: 10
2017-11-05 12:15:09,218 -- DEBUG -- Argument 2: 12
2017-11-05 12:15:09,218 -- DEBUG -- Argument 3: 23
2017-11-05 12:15:09,218 -- INFO -- Le résultat est 45
2017-11-05 12:15:09,218 -- DEBUG -- apres
```

Iterateurs

Itérateurs

Le parcours d'itérables

Pour rappel, la boucle `for` permet de parcourir un objet **itérable**.

```
for i in range(10):  
    print(i)  
  
for elem in ma_liste:  
    print(elem)  
  
for lettre in "hello":  
    print(lettre)
```

Itérateurs

Le parcours d'itérables

Pour savoir si un objet est itérable :

```
try:
    iter(mon_objet)
except TypeError:
    # non itérable
else:
    # itérable
```

Autre méthode avec `isinstance` :

```
import collections
if isinstance(mon_objet, collections.Iterable):
    # itérable
else:
    # non itérable
```

Itérateurs

Le parcours d'itérables

La boucle `for` peut être réalisée avec le code suivant :

```
it = iter(mon_iterable)
while True:
    try:
        next(it)
    except StopIteration:
        break
```

Un objet itérable doit implémenter deux méthodes :

- `__iter__(self)` : Doit retourner l'objet itérateur
- `__next__(self)` : Doit retourner l'élément suivant à parcourir, ou lever une exception `StopIteration` si tous les éléments ont été parcourus.

Itérateurs

Créer un objet itérable

```
class MyRange():  
    def __init__(self, max_val):  
        self.max_val = max_val  
        self.i = 0  
  
    def __iter__(self):  
        return self  
  
    def __next__(self):  
        if self.i < self.max_val:  
            i = self.i  
            self.i += 1  
            return i  
        else:  
            raise StopIteration()
```

Un **générateur** est une **fonction** qui se comporte comme un itérateur.

C'est un raccourci syntaxique pour créer un itérateur.

```
def range_gen(max_val):  
    i = 0  
    while i < max_val:  
        yield i  
        i += 1
```

Itérateurs

Générateurs

- Quand une fonction générateur est appelé, son code n'est **pas exécuté** ! C'est uniquement le générateur qui est retourné.
- A l'appel de la fonction `next()` , le code est exécuté jusqu'au mot clé **yield** qui retourne l'objet courant et sauvegarde l'état de la fonction.
- A l'appel suivant de `next()` , le code reprend au `yield` où il s'était arrêté.
- Ils peuvent être utilisés par toutes les fonctions pouvant **consumer** un itérateur.

```
for elem in squares_gen:  
    elem
```

Ou encore :

```
sum(squares_gen)
```

Il est possible de créer des générateur avec une syntaxe similaire aux lists comprehensions.

```
squares_gen = (i*i for i in range(10))
```

Note :

Puisque les parenthèses sont utilisées pour créer un générateur, pour faire un tuple en compréhension il faut explicitement écrire `tuple` .

```
tuple(i*i for i in range(10))
```


Multi-threading

Multi-threading

module threading

Le module 'threading' permet d'exécuter des tâches de façon concurrentes.

- Mémoire partagée
- Ordonné par le module threading
- Création rapide et peu coûteuse en mémoire
- Un seul cœur utilisé

Multi-threading

module threading

Exemple de création et d'exécution d'un thread :

```
import threading

class MonThread(threading.Thread):
    def run(self):
        # Thread code...

for i in range(3):
    t = MonThread()
    t.start()
```

Multi-threading

module threading

Il est possible de créer directement l'instance du Thread sans devoir explicitement créer une classe :

```
def compute(arg1, arg2):  
    # Thread code...  
  
for i in range(5):  
    threading.Thread(target=compute, \  
                    args=(arg1, arg2,)).start()
```

Multi-threading

module multiprocessing

Le module '**multiprocessing**' permet de créer et d'exécuter des **processus**.

- Création similaire au Thread
- Aucune mémoire partagée
- Ordonné par l'OS
- Création plus lente et coûteuse que threading
- Multi-cœur possible

Multi-threading

module multiprocessing

```
import multiprocessing

class MonProcess(multiprocessing.Process):
    def run(self):
        # Process code...

for i in range(3):
    t = MonProcess(i)
    t.start()
```

En programmation concurrente il faut être particulièrement vigilant, notamment lors des accès aux **ressources partagées** et des phases de **synchronisations**.

Outils usuels disponibles :

- Verrous
- Mutex
- Sémaphores
- ...

Multi-threading

Queue

Le module '`queue`' implémente un paradigme multi-producteur / multi-consommateur thread-safe.

Il est recommandé pour transmettre des informations entre les différents thread.

Le module définit les 3 classes suivantes :

- `Queue` : Queue FIFO
- `LifoQueue` : Queue LIFO
- `PriorityQueue` : tuple (priority, data) avec la priorité la plus faible retournée en premier

Multi-threading

Queue

Définition d'une classe 'worker' qui va traiter un certain nombre de tâche qui sont extraite de la queue.

```
1 class WorkerQueue(threading.Thread):
2     def __init__(self, queue):
3         super().__init__()
4         self.queue = queue
5
6     def run(self):
7         while True:
8             station = self.queue.get()
9             if station is None:
10                 break
11             # Task code...
12             self.queue.task_done()
```

Multi-threading

Queue

Création des tâches et affectation aux différents 'workers'.

```
1 q = queue.Queue()
2 workers = []
3 for i in range(worker_count):  # création des workers
4     worker = DepartQueue(q, res)
5     worker.start()
6     workers.append(worker)
7
8 for station in stations:  # Ajout des tâches
9     q.put(station)
10
11 q.join()  # attente que toutes les tâches soient terminées
12
13 for i in range(worker_count):  # arrêt des workers
14     q.put(None)
15 for worker in workers:
16     worker.join()
```

Futures

Le module '`concurrent.futures`' propose un plus haut degré d'abstraction pour exécuter des algorithmes `asynchrones` à l'aide de la classe abstraite `Executor`.

Les algorithmes asynchrones peuvent être exécutés :

- à l'aide de `threads concurrents` en utilisant la classe `ThreadPoolExecutor`,
- en parallèle à l'aide de processus en utilisant la classe `ProcessPoolExecutor`.

Exemple :

```
from concurrent.futures import *  
  
with ThreadPoolExecutor(max_workers=2) as executor:  
    f1 = executor.submit(sum, range(50_000_000))  
    f2 = executor.submit(sum, range(50_000_000))  
  
    print(f1.result() + f2.result())
```

Aucun parallélisme n'est appliqué => aucun gain

Exemple :

```
from concurrent.futures import *

with ProcessPoolExecutor(max_workers=2) as executor:
    f1 = executor.submit(sum, range(50_000_000))
    f2 = executor.submit(sum, range(50_000_000))

    print(f1.result() + f2.result())
```

Deux processus effectuent les tâches **en parallèle** => speedup ~ 2

Exemple de la documentation officielle

(<https://docs.python.org/3/library/concurrent.futures.html>) :

```
from concurrent.futures import *
import math

PRIMES = [
    112272535095293,
    112582705942171,
    112272535095293,
    115280095190773,
    115797848077099,
    1099726899285419]

with concurrent.futures.ProcessPoolExecutor() as executor:
    for number, prime in zip(PRIMES, executor.map(is_prime, PRIMES)):
        print(f"{number} is prime: {prime}")
```

Interfaces graphiques

Framework cross-plateform d'interfaces graphiques en Python :

- wxPython
- PyGTK
- PyQt
- Kivy
- TkInter

Paradigme de programmation évènementielle.

Une application TkInter est composée d'une ou plusieurs fenêtres elles mêmes composées d'éléments graphiques appelés Widgets . Ces éléments sont agencées selon une structure hiérarchique.

La structure de la librairie a changée entre Python 2 et Python 3 :

Python 2	Python 3
Tkinter	tkinter
tkFileDialog	tkinter.filedialog
tkMessageBox	tkinter.messagebox
...	

Pour s'assurer que la librairie fonctionne :

```
>>> import tkinter
>>> tkinter._test()
```

TkInter

Exemple d'utilisation

Exemple minimaliste :

```
import tkinter as tk

root_window = tk.Tk()

label = tk.Label(root_window, text="Hello World")
label.pack()    # Geometry manager

root_window.mainloop()
```

Les **Widgets** sont des éléments graphiques à afficher.

Quelques Widgets de base :

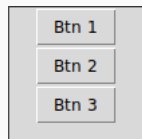
- **Label** : Afficher du texte
- **Button** : Action de l'utilisateur
- **Entry** : Saisie clavier d'un champs texte
- **Canvas** : Espace de dessin
- **Frame** : Permet de contenir d'autres Widgets

TkInter

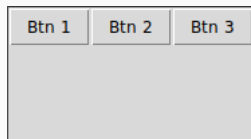
Pack geomerty manager

Le `geometry manager` permet d'agencer des widgets les uns par rapport aux autres.

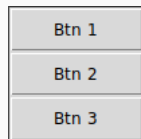
```
mon_widget.pack(side=, fill=, expand=)
```



side=tk.TOP



side=tk.LEFT



fill=tk.BOTH

Pour lier une action à un bouton, il faut affecter une méthode à l'option **command**.

```
btn["command"] = ma_fonction  # sans les parenthèses!  
  
# qui est équivalent à  
btn.config(command=ma_fonction)
```

Tkinter

Exemple avec classe

```
1 import tkinter as tk
2 from tkinter import messagebox
3
4 class MainFrame(tk.Frame):
5     def __init__(self, master=None):
6         super().__init__(master)
7         self.pack()
8         self.button = tk.Button(self)
9         self.button["text"] = "Click"
10        self.button["command"] = self.hello
11        self.button.pack()
12
13    def hello(self):
14        messagebox.showinfo("Message:", "Hello")
15
16 root = tk.Tk()
17 app = MainFrame(master=root)
18 app.mainloop()
```