6- La boucle with (avec) : lecture /écriture d'un fichier

Cette boucle permet de lire un fichier rapidement.
En quittant with, le fichier est fermé automatiquement de façon propre.

```
#Afficher le fichier
#en quittant with, le fichier va être fermé de façon propre (ne peut plus être lu)
with open("main.py", "r") as fichier:
    print(fichier.read())

print("Bonjour") #la sortie sera juste Bonjour, car le fichier a été fermé par le with
print(fichier.read()) #ligne inutile car le fichier a été fermé


#Ecrire dans un fichier puis le lire
with open("sortie", "w") as fichier:
    print(fichier.write("Bonjour"))

with open("sortie", "r") as fichier:
    print(fichier.read())
```

Exemple : lire /écrire dans un fichier avec with

1- Ouvrir, lire et écrire dans un fichier : les modes

open(.), read(.), readline(.), seek(.), tell(), close(), write()

Les différents modes

On a les modes suivants :

Mode	Action	Commentaire
r	Read	*Lecture seule.*
w	Write	*Ecriture seule.* Le contenu du fichier est perdu.
r+	Read et Write	*On peut lire et écrire.* Pour l'écriture, le contenu du fichier est perdu.
a	Read et Write	*On peut lire et écrire*. L'écriture ajoute à la fin di fichier (append).
b	Binaire	Lire des fichiers qui ne sont pas purement du texte.
rb	Read et binaire	Lire en lecture seule + fichier binaire.

2- Ouvrir et lire un fichier

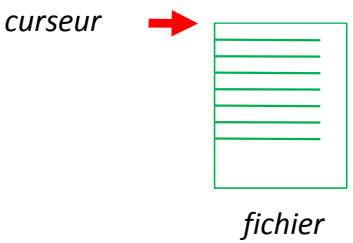
open(.), read(.), readline(.), seek(.), tell(), close()

Etapes pour lire un fichier

a- ouvrir avec *open(chemin, mode)* : on aun pointeur **p**.
b- lire avec ***p***.*read(nbChar)* avec ***p***.*readline(nbChar)*
c- fermer le pointeur avec ***p***.*close()*

Notion de curseur

a- un curseur est positionné en début du fichier.
b- *read()* déplacé le curseur jusqu'à la fin.
c- pour rembobiner, utiliser ***p***.*seek(position)*
d- obtenir la position du curseur: ***p***.*tell()*




```python
#Lire un fichier avec read()
fichier = open("montexte.text","r")
texte = fichier.read()
print(texte)

#Relire le fichier en remontant le curseur
fichier.seek(0)
texte1 = fichier.read()
print(texte1)

#Méthodes readline(.), tell(.) et seek(.)
fichier.seek(0)
texte2 = fichier.readline() #lit une ligne
print(fichier.tell()) #position curseur
texte3 = fichier.read(10) #lit 10 caractères
print(fichier.tell())

#Fermer le fichier
fichier.close()
```

Code : lecture d'un fichier

3- Ouvrir et lire un fichier avec les boucles for et with

open(.), read(.), readline(.), seek(.), tell(), close()

Avec la boucle for

for lit automatique le fichier ligne à ligne car l'objet fichier est un objet itérable. C'est la méthode de lecture la plus simple.

Avec la boucle with

with lit le fichier (et écrit dans le fichier) tout en le fermant automatiquement (proprement) à la fin du with.

Vous ne pouvez donc plus travailler sur un fichier traité avec **with** : il faut le ré ouvrir.

with permet aussi d'écrire dans un fichier.

```
#Lire un fichier avec for et with
fichier = open("montexte.text","r")
for ligne in fichier:
    print(ligne, end="")

#Lire un fichier avec with
fichier.seek(0)
with open("montexte.text", "r") as fichier:
    print(fichier.read())
```

Code : lecture d'un fichier

4- Ouvrir et écrire dans un fichier

open(.), close(), write()

Etapes pour écrire dans le fichier

a- ouvrir avec *open(chemin, mode)* : on obtient un pointeur **p**.
b- écrire avec ***p***.*write(chaine)*
c- fermer le pointeur avec ***p***.*close()*

```
#Ecrire dans un fichier
fichier = open("montexte.text","w")
texte = fichier.write("Joli texte")
print(texte)
```

Code : écrire dans un fichier

```
#Ecrire dans un fichier avec
with open("montexte.text", "w") as fichier:
    print(fichier.write("Bonjour"))
```

Code : écrire dans un fichier (boucle with)

Nota

a- avec la boucle **with** qui ferme automatiquement le fichier à lire/écrire.
b- Pour des fichiers binaires (word, excel, etc.) , utiliser des composants spécialisés.

5- Parcourir le système de fichiers

Le module pathlib permet de créer un objet qui représente un dossier et de parcourir ce dossier. *Path(.)* permet de créer des chemins. *iterdir()* permet de faire le parcours du dossier.

```
# Importation (from pathlib import Path)
import pathlib
mondossier = pathlib.Path("/home/toto") #définir un dossier

# Parcourir le dossier et obtenir le chemin des fichiers
for chemin in mondossier.iterdir():
    fichier = open(str(chemin))
```

Code : exemple utilisation pathlib

5- Parcourir le système de fichiers : suite

Path(.) crée des chemins, *iterdir()* permet de faire le parcours du dossier, *is_dir()* détermine si on a un dossier.

```
# Liste sous dossiers
>>> p = Path('.') # dossier parent
>>> [x for x in p.iterdir() if x.is_dir()])
>>> list(p.glob('**/*.py')))  # Fichiers Python : recherche avec glob

# Naviguer dans l'arborescence
>>> p = Path('/etc')
>>> q = p / 'init.d' / 'reboot'
>>> q
PosixPath('/etc/init.d/reboot')
>>> q.resolve()
PosixPath('/etc/rc.d/init.d/halt')

# Propriétés des chemins
>>> q.exists()
True
>>> q.is_dir()
False
```

Code : exemple utilisation pathlib

1- Caractéristiques des fonctions

- Permettent de factoriser du code, se retrouvent dans des modules
- Reçoivent des données en entrée et retournent d'autres données (boîte noire)
- Syntaxe simple (sans accolades), pas de typage explicite
- Ce sont des objets comme d'autres

```
def nomfonction(paramètres) :
    inst1
    inst2
      .
      .
    instn
    return quelquechose
```

Syntaxe

2- Création de fonctions en Python

a- Une fonction simple

```
# Une fonction simple
def direbonjour(nom):
    return "Bonjour" + " " + nom

# utiliser la fonction
print(direbonjour("Dupont"))
```

Code : fonction simple (main.py)

b- Une fonction est un objet

```
# Une fonction comme un objet
# myfonction est une fonction passe en paramètre
def appeler(myfonction, nom):
    return myfonction()

# Utiliser la fonction
print(appeler(direbonjour, "Dupont"))
```

Code : une fonction en paramètre

2- Création de fonctions en Python : suite

a- Une fonction dans une autre

Pour mieux organiser son code, il est possible de placer des fonctions dans une fonction comme dans l'exemple.

```
# Une fonction dans une autre
def remercier(nom):
    def direbonjour(nom):
        return "Bonjour" + " " + nom

    return direbonjour(nom)

# utiliser la fonction
print(remercier("Hélène"))
```

Code : fonction interne (main.py)

3- Les paramètres des fonctions

a- Paramètres positionnels

Tous les paramètres obligatoires (positionnels) indiqués dans la parenthèse **doivent** être donnés au moment de l'appel, et dans le même ordre.

```
# Fonction avec paramètres positionnels
def direbonjour(prenom, nom):
    return "Bonjour" + " " + prenom + " " + nom

# Utiliser : 2 arguments attendus
print(direbonjour("Jean", "Dupont"))
```

fonction avec 2 paramètres

b- Paramètres nommés

Certains paramètres peuvent être nommés.

```
# Fonction avec paramètres nommés
def direbonjour(prenom, nom):
    return "Bonjour" + " " + prenom + " " + nom

# Utiliser : 2 arguments attendus
print(direbonjour(nom = "Dutour", prenom = "Jean"))
```

fonction avec 2 paramètres nommés

3- Les paramètres des fonctions : suite

a- Mauvaise utilisation des paramètres

Le code ci-dessous sera KO : parce qu'à l'appel, il faut d'abord faire figurer les paramètres obligatoires (positionnels).

```
# Fonction avec paramètres nommés
def direbonjour(prenom, nom):
    return "Bonjour" + " " + prenom + " " + nom

# Appel mauvais : les paramètres obligatoires d'abord
print(direbonjour(prenom = "Jean", "Dutour"))
```

fonction mal appelée KO

SyntaxError : positional argument follows keyword argument

4- Les paramètres avec valeur par défaut

Donner des valeurs par défaut

```
# Paramètres avec valeurs par défaut
def direbonjour(prenom = "Jean", nom = "Dutour"):
    return "Bonjour" + " " + prenom + " " + nom

# Appel sans fournir de paramètre
print(direbonjour())
```

← *Code : 2 valeurs par défaut*

```
# Paramètres avec 1 valeur par défaut
def direbonjour(prenom, nom = "Dutour"):
    return "Bonjour" + " " + prenom + " " + nom

# Appel sans fournir de paramètre
print(direbonjour(prenom = "Jean"))
```

← *Code : 1 valeur par défaut*

SyntaxError: non-default argument follows default argument

```
# Paramètres avec 1 valeur par défaut
def direbonjour(prenom = "Jean", nom):
    return "Bonjour" + " " + prenom + " " + nom

# Appel mauvais : les paramètres obligatoires d'abord
print(direbonjour("Dutour"))
```

← *KO : les paramètres obligatoires d'abord*

5- Les arguments variables

On peut avoir un nombre d'arguments variable en utilisant comme paramètres **args* (***arguments***) et ***kwargs* (***key words arguments*** = arguments nommés).

On peut avoir autant d'arguments qu'on veut sans modifier la signature de la fonction en question.

```
# Arguments variables
def direbonjour(*args, **kwargs):
    return "Bonjour " + ", ".join(args)

# Appel avec 2 arguments
print(direbonjour("Jean","Dutour"))
```

Appel avec des arguments obligatoires

```
# Arguments variables
def direbonjour(*args, **kwargs):
    return "Bonjour " + ", ".join(args) + " " + kwargs["nom"]

# Appel avec 2 arguments
print(direbonjour("Jean","Jacques", nom ="Goldman"))
```

Arguments obligatoires et nommés

6- Variables globales, variables locales

Variables locales

On peut créer des variables au sein d'une fonction qui ne seront pas visibles à l'extérieur de celle-ci ; on les appelle **variables locales.**

```
>>> def mafonction():
...     x = 2
...     print 'x vaut',x,'dans la fonction'
...
>>> mafonction()
x vaut 2 dans la fonction
>>> print x
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'x' is not defined
```

Variables locales

Nota : Si on veut modifier une variable globale dans une fonction, il faut utiliser le mot-clé **global**.

```
>>> def mafonction():
...     global x
...     x = x + 1
...
>>> x=1        <-- variable globale
>>> mafonction()
>>> x
2
```

Une variable passée en argument est considérée comme **locale lorsqu'on arrive dans la fonction.**

7- Documenter une fonction et un module, help(.)

On peut facilement documenter une fonction ou un module avec une **docstring** et profiter de la documentation automatique de Python.

Fonction documentée

```
#Une fonction documentée
def direbonjour(prenom, nom):
    """ Cette fonction vous fait Coucou. """
    return "Bonjour" + " " + prenom + " " + nom

# Appel
print(direbonjour("Jean", "Dutour"))
```

docstring à placer juste après la 1ère ligne.

Python place la **docstring** dans la variable spéciale **__doc__** associée à l'objet fonction comme étant l'un de ses attributs (voir POO).

print(direbonjour.__doc__) affichera la doctring.

Scruter les fonctions avec help(.)

```
>>>help(direbonjour)
```

```
Help on function moyenne in module __main__:

direbonjour(prenom, nom)
    Cette fonction vous fait Coucou.
(END)
```

7- Documenter une fonction et un module, help(.) : suite

Scruter les fonctions avec help(.)

Help() fonctionne sur n'importe quelle fonction d la librairie standard.

```
>>>help(len)
```

```
Help on built-in function len in module builtins:

len(obj, /)
    return the number of items in a container.
(END)
```

```
>>>help(print)
```

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
(END)
```

7- Documenter une fonction et un module, help(.) : suite 1

Afficher le prompt d'aide avec help(.)

help() sans paramètre affiche un nouveau prompt permettant d'avoir de l'aide sur n'importe quel sujet.

help>

Instruction	Commentaire
int	Aide sur les entiers
float	Aide sur les floats
topics	Aide classée par sujets. Chaque sujet fournit un tutoriel simple.
q	Quitter le prompt d'aide

Demander de l'aide dans le prompt aide.

7- Documenter une fonction et un module, help(.) : suite 2

Enregistrer votre fichier en .py (module) + documenter

En sauvegardant votre code dans des fichiers .py, vous créez des modules. Utilisez aussi un docstring pour documenter les modules.

direbonjour.py

```
"""
    Un module tout neuf et cool chargé
    des salutations.
"""
def direbonjour(prenom, nom):
    """ Cette fonction vous fait Coucou. """
    return "Bonjour" + " " + prenom + " " + nom

# Appel
print(direbonjour("Jean", "Dutour"))
```

Module direbonjour *documenté*

```
>>>import direbonjour
>>> direbonjour.direbonjour("Jean", "Dutour"))
Bonjour Jean Dutour
```

Importer votre module

```
>>>import direbonjour
>>>help(direbonjour)
```

```
Help on module direbonjour:

NAME
    direbonjour

DESCRIPTION
    Un module tout neuf et cool chargé
    des salutations.

FUNCTIONS
    direbonjour(prenom, nom)
        Cette fonction vous fait Coucou.

FILE
        /direbonjour.py
(END)
```

Aide du module direbonjour

7- Documenter une fonction et un module, help(.) : suite 3

PYTHONPATH

Python récupère les modules importés en regardant dans la liste path du module sys contenant tous les chemins d'importation possibles.

```
>>>import sys
>>>sys.path
>>>import pathlib
```

Les chemins sont recherchés dans la liste sys.path

Ajouter au path le chemin vers le dossier du .py

Vous pouvez ajouter vous-même le chemin de votre module dans le **path** avec la méthode *append()* qui ajoute un élément à la liste.

```
>>>sys.path.append(' /chemin/vers/le/dossier/de/votre/module')
```

8- Unpacking sur les itérables

Importance de l'itération

L'itération est très importante en Python.
On peut itérer sur n'importe quel itérable chaque fois que cela a un sens : liste, tuple, set, str, …

```
>>>max([5,8,16])
16
>>>max((5,8,16))
16
>>>max(set([5,8,16]))
16
```

La fonction max fonctionne pour ces 3 itérables

L'unpacking (déballage) est un raccourci qui consiste à abréger la notation lors de la récupération des éléments d'un itérable, dans des boucles ou dans des fonctions.

8- Unpacking sur les itérables : suite

Exemples d'unpacking

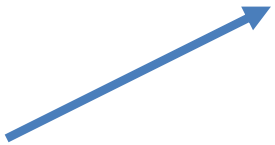
```
>>>reponses = ['Oui','Non']
>>>a = reponses[0]
>>>b = reponses[1]
>>>a
'Oui'
>>>b
'Non'
```

Récup de la liste

```
>>>(a, b) = ['Oui','Non']
>>>a
'Oui'
>>>b
'Non'
```

Déballage de la liste

Unpacking sur le générateur range(.)



```
>>>(a, b, c) = range(3)
>>>a
0
>>>b
1
>>>c
2
```

Déballage du range()

8- Unpacking sur les itérables : suite 1

Exemples d'unpacking

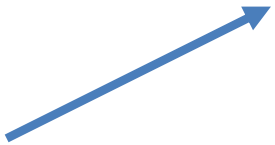
```
>>>a = "mol"[0]
>>>b = "mol"[1]
>>>a
'm'
>>>b
'o'
```

Récup des caractères

```
>>>(a, b, c) = "mol"
>>>a
'm'
>>>b
'o'
>>>c
'l'
```

Déballage chaine

On omet les parenthèses sur le tuple.



```
>>>a, b, c = "mol"
>>>a
'm'
>>>b
'o'
>>>c
'l'
```

Déballage chaine

8- Unpacking sur les itérables : suite 2

Exemples d'unpacking

Mettre * quand il y a trop de choses à déballer par rapport au nombre d'éléments du tuple.

```
>>>a, b, *c = range(4)
>>>a
0
>>>b
1
>>>c
[2,3]
```

Déballage range()

```
dico = {"Hello":"Bonjour", "Bye":"Au revoir",}
for en, fr in dico:
    print(en + "" + fr)
```

Parcours d'un dictionnaire

8- Unpacking sur les itérables : suite 3

Unpacking sur les paramètres des fonctions avec * ou **

```python
# Fonction appelée sans unpacking
def somme(a = 0, b = 0):
    return a + b

data = (5,10)
resultat = somme(data[0], data[1])
print(resultat) # donne 15
```

Appel sans unpacking

```python
# Fonction appelée avec unpacking
def somme(a = 0, b = 0):
    return a + b

data = (5,10)
resultat = somme(*data)
print(resultat) # donne 15
```

*Appel avec unpacking : on passe le tuple (notez le *)*

```python
# Fonction appelée sans unpacking (dictionnaire)
def somme(a = 0, b = 0):
    return a + b

dicodata = {"a": 5, "b": 10)
resultat = somme(dicodata['a'], dicodata['b'])
print(resultat) # donne 15
```

Appel sans unpacking, en passant les valeurs du dico

```python
# Fonction appelée avec unpacking (dictionnaire)
def somme(a = 0, b = 0):
    return a + b

dicodata = {"a": 5, "b": 10)
resultat = somme(**data)
print(resultat) # donne 15
```

*Appel avec unpacking : on passe le dico directement (notez le double **)*

8- Unpacking sur les itérables : suite 4

Unpacking sur les valeurs de retour

```python
# Retour d'une fonction sans unpacking
import random

def random_point(start = 0, end = 0):
    x = random.randint(start, end)
    y = random.randint(start, end)
    return (x,y)

# Appel : p1 est un tuple
p1 = random_point()
```

Retour d'un tuple sans unpacking

```python
# Retour d'une fonction sans unpacking
import random

def random_point(start = 0, end = 0):
    x = random.randint(start, end)
    y = random.randint(start, end)
    return (x,y) # ou return x,y plus simplement

# Appel
x, y = random_point()
```

Retour d'un tuple avec unpacking

On a l'illusion de retourner 2 valeurs, alors qu'on retourne un tuple.

On peut utiliser le 1er élément du tuple retourné comme **x**, et le 2ème comme **y**.

On peut continuer à utiliser le tuple comme ceci **x, y** ou comme ceci **(x, y)**.

9- Listes, dictionnaires et sets en intension

Une liste en intension permet de générer des listes en compréhension (cf. maths - ensemble en compréhension)

```
# Liste explicite de carrés
# [0, 1, 4, 9, 16 ,25 , 36, 49 ,64 ,81]
nombres = range(0,10)
carres = []
for x in nombres:
    carres.append(x * x)
```

Liste explicite (4 lignes)

```
# Liste carres de carrés en intension
carres = [x * x for x in range(10)]
```

Liste en intension (1 seule ligne)

```
# Obtenir des carrés dans nombres
nombres = range(0,10)
carres = [x * x for x in nombres]
nombres = carres
```

Liste en intension (3 lignes)

```
# Obtenir des carrés dans nombres
# (en plus court)
nombres = range(0,10)
nombres = [x * x for x in nombres]
```

Liste en intension (2 lignes

9- Listes, dictionnaires et sets en intension : suite

Filtrer dans la liste en compréhension

```
# Filtrage explicite avec un for
# Obtenir [0, 4, 16, 36, 64]
nombres = range(10)
carres = []
for x in nombres:
    if x % 2 == 0:
        carres.append(x * x)
```

Liste explicite (5 lignes) de nombres pairs

```
# Filtrage avec liste en intension
# Obtenir [0, 4, 16, 36, 64]
nombres = [x * x for x in range(10) if x % 2 == 0]
```

Liste en intension (1 seule ligne)

```
nombres = range(10)
carres = []
for x in nombres:
    if x % 2 == 0:
        carres.append(x * x)
```

```
carres = [x * x  for x in nombres if x % 2 == 0]
```

9- Listes, dictionnaires et sets en intension : suite 1

Dictionnaires et sets en intension

```
# Filtrage avec liste en intension
# Obtenir [0, 4, 16, 36, 64]
nombres = [x * x for x in range(10) if x % 2 == 0]
```

Liste en intension

```
# Dictionnaire et set en intension
diconombres = {"x": x * x for x in range(10) if x % 2 == 0}
```

Dictionnaire en intension

```
# Set en intension
# Obtenir {0, 4, 16, 36, 64}
setnombres = {x * x for x in range(10) if x % 2 == 0}
```

Ensemble (set) en intension

1- Créer un décorateur

Un décorateur est une fonction destinée à modifier le comportement d'une autre fonction. On dit qu'elle décore la fonction en question (notion typiquement Python).

Le décorateur est appelé comme ceci : @nomdecorateur.

```
# Une fonction qui va décorer une autre
def decorateur(fonction):
    def imposteur(nom): #mêmes arguments que la fonction à décorer
        print("Appel de la fonction " + fonction.__name__ + " avec le paramètre " + nom)
        return fonction(nom)

    return imposteur

@decorateur                    <-- Décore la fonction direbonjour(.)
def direbonjour(nom):
    return "Bonjour" + " " + nom

@decorateur                    <-- Décore la fonction remercier(.)
def remercier(nom):
    return "Merci" + " " + nom

print(direbonjour("Dupont"))
print(remercier("Dolto"))
```

2- Intérêt d'un décorateur

- Récolter des informations sur les appels de fonctions.
- Ajouter des fonctionnalités/infos additionnelles à un groupe de fonctions.
- Journaliser les appels d'une fonction.
- Eviter les répétitions.

Décorateurs pré-définis (voir POO)

@staticmethod :
pour déclarer une méthode de classe (ou statique)

@property : décorateur de propriété = la fonction devient un attribut.

Nota : Plusieurs décorateurs peuvent être appliqués à une fonction.

1- Un générateur

Le générateur est une « sorte de fonction » qui a un potentiel de générer des données. Ces données ne sont pas générées immédiatement après la création du générateur.

Mécanisme de génération

* On applique la méthode *next()* (*__next__()*) au générateur : une donnée est générée.
* On applique encore *next()*, la donnée suivante est générée et la précédente oubliée.
* Et ainsi de suite.

Nota : - Le générateur ne sert pas à stocker des données persistantes.
- On n'applique pas soi-même la méthode *next()*.

Créer un générateur

Deux moyens :

1- Utiliser une liste en intension

2- Utiliser la notation des fonctions, avec une clause yield.

2- Générateur avec une liste en intension

On peut créer un générateur à partir d'une liste en intension.
L'objet obtenu n'est pas une fonction, ni une liste.

```
# g est un générateur, pas une liste
g = (x * x for x in range(10000) if x % 2 == 0)

# Lire le générateur g, générer une 1ère valeur a
a = next(g)

# Lire le générateur g, générer une 2ère valeur b
# La 1ère est oubliée par le générateur
b = next(g)
```

Un générateur

Un générateur est itérable

On peut donc le parcourir avec for.
open() et *range()* sont des générateurs : on peut leur appliquer *next()*. For gère le *next()* tout seul.

L'exception StopIteration est levée si on est à la fin du générateur. La boucle for gère tout seul StopIteration.

```
# Ils sont itérables
for x in g:
    print(x)
```

```
# Créer une liste à partir
# du générateur
liste = tuple(g)
```

Obtenir une liste

3- Générateur avec yield

On peut créer un générateur « comme s'il s'agissait » d'une fonction.
On utilise le mot clé yield pour indiquer les ordres/étapes de génération à exécuter.

yield retourne le résultat mais ne termine pas l'exécution du générateur (elle est juste mise en pause)

```
#Génerer 1, 2, 3
def generateur():
    yield 1
    yield 2
    yield 3

#initialiser le générateur
gen = generateur()
print(gen.__next__())
print(gen.__next__())
print(gen.__next__())
print(gen.__next__()) #StopIteration
```

Un générateur avec yield

```
# Génère 4 fois le chiffre 2
def generateur():
    while True:
        yield 2

gen = generateur()
print(gen.next())
print(gen.next())
print(gen.next())
```

Un générateur avec yield

3- Générateur avec yield : suite

Puisque c'est « comme s'il s'agissait » d'une fonction, on peut du coup passer des paramètres au générateur.

```
# Un générateur pour yielder
# des carrés
def carres():
    for x in range(100):
        yield x * * 2

# carres est un générateur,
# pas une fonction
g = carres()
for x in g:
    print(x)
```

Générateur sans paramètres

```
# Générateur carres()avec
# transmission des paramètres
def carres(start = 0, end = 100):
    for x in range(start, end):
        yield x * * 2

maliste  = list(carres(1, 3))
montuple = tuple(carres(1, 3))
```

Générateur avec paramètres

On peut **générer une liste ou un tuple à la demande**.

4- Générateur : intérêt

a- Générer des données à la demande

b- Enfermer des traitements complexes dans le générateur

c – Optimisation de la mémoire

d- Obtenir rapidement des structures complexes : listes, tuples.

e- etc.

1- Fonctions lambda

a- Ce sont des fonctions simples définies en ligne (inline)

b- Elles sont définies sur une seule ligne

c- Il n'y a pas de mot clé def.

```
# Une fonction lambda
ajouter1 = lambda x, y : x + y

# une fonction ordinaire équivalente
def ajouter2(x,y):
    return (x+y)

print(ajouter1(5,10))
print(ajouter2(5,10))
```

Exemple : fonction lambda

Nota : ne pas en abuser.

2- Fonctions lambda

1- Gestion des erreurs : les objets de type Exception

La gestion des erreurs permet d'éviter que votre programme plante. Les erreurs peuvent être gérées en utilisant un bloc try/except et en manipulant des objets renvoyés par le programme : les objets Exception.

Opération	Erreur	Objets
>>> 1/0	Division par 0	ZeroDivisionError
>>> 2 + "15"	Types incompatibles	TypeError
>>> var	Nom de variable inconnu	NameError
>>> liste = [] >>> liste [0]	La liste ne contient rien	IndexError

>>> d = {}
>>> d={b:1}
>>> d{0}
	La clé 0 n'existe pas	KeyError

Objets de type Exception

Nota : Ces objets retournés peuvent être manipulés.

Peut être une classe définie par vous-même (voir POO).

Générer une erreur

```
# Générer une erreur
if n == 1:
    raise TypeError("Le type testé est mauvais.")
```

Gérer l'erreur

On dispose de plusieurs mots clés : try, except, else, finally pour gérer les erreurs.

2- Gérer une exception

Si l'exécution du try échoue, on attrape l'erreur dans le except : c'est dans except qu'on fait la gestion de l'erreur.

```
# Gérer la division par 0
>>> try:
...     n = 0
...     b = 8/n
... except ZeroDivisionError:
...     print("Division impossible")
Division impossible
```

Division par 0

```
# Utiliser else et finally
try:
    n = 0
    b = 8/n
except ZeroDivisionError:
    print("Division impossible")
else:
    print("OK")
finally:
    print("Toujours exécuté")
```

Division par 0

```
# Gérer l'ouverture de fichier
try:

    fichier = open(chemin)

except Environment as e:
    #Problème
    print("Ouverture KO", e)
else:
    #Aucun souci
    print(fichier.read())
finally:
    #Toujours exécuté
    try:
        fichier.close()
    except NameError:
        pass
```

Lire l'objet

Ouverture d'un fichier

Surtout utilisé pour le nettoyage. **with** nettoie tout seul : en ouvrant le fichier avec **with**, ce bloc sera inutile.

2- Gérer une exception : suite

On peut avoir plusieurs blocs **except** et propager les erreurs avec **raise**.

```
# Plusieurs except
dico = {'age':1, 'poids':50}
try:
    print(dico('taille'))

except KeyError:
    #Problème, définir l'élément
    dico['taille'] = 92

    n = 0
    div = 8/n # y aura erreur ici
except:
    print("Autre erreur")

else:
    print("Aucun souci")

finally:
    print("Toujours exécuté")
```

KeyError

On ne saura pas quelle erreur sera générée (mauvaise pratique) : il faut toujours paramétrer le **except** ou propager (remonter) les exceptions avec **raise**. Ou raise Exception(.)

raise ou **raise** Exception("Division par 0 ")

Exception personnalisée : il peut s'agit de votre propre classe Maclasse qui hérite de la classe Exception ou BaseException (voir POO).