j6

J6: Objectifs

- Portée des noms
- Scope
- Local
- global
- non local
- Liste d'appels
- arguments positionnels
- arguments nommés
- arguments en nombre variable
- valeurs par défaut
- argument fonctionnel
- compacter/décompacter *, **
- zipper/dézipper
- modules
- packages

- lambda
- fonction anonymes
- générateurs
- yield
- decorator
- style "fonctionnel"

Call by object reference

Question : Une fonction peut-elle modifier ses arguments ?

- Les autres langages utilisent
  - *Call by value* : on évalue les arguments, avant de passer les valeurs à la fonction. Elle ne peut pas changer les arguments. (Langage C)
  - *Call by reference* : les adresses des arguments sont transmises. La fonction peut modifier les informations à ces adresses (Fortran , VB, PHP, C++/C#)
  - *Call by name* et +…

- Python est "*Call by object reference*" ou "*Call by sharing*"
  - Pour des types non modifiables (int, float, str, tuple , cela s'apparente au "Call by value")
    ➔ La fonction ne peut pas modifier de tels arguments
  - Pour les autres, au "call by reference"
    ➔ La fonction peut modifier la valeur de ces arguments

- Attention des types non modifiables peuvent contenir des références à des objets modifiables et des effets de bord sont possibles malgré les apparences.

Résultat d'une fonction

- pas de valeur
  absence de return

```
None
```

- une valeur

```
return x*math.sin(x)
…
print(f(3))
```

- plusieurs valeurs
  on récupère un tuple

```
return 2*pi*r, pi*r**2
…
périmètre, surface = cercle(1.5)
```

- Emploi de la variable _ pour indiquer que l'une des valeurs ne nous intéresse pas

```
périmètre, _ = cercle(1.5)

_, surface = cercle(1.5)
```

Arguments positionnels ou nommés

- Notion qui intervient dans l'utilisation de la fonction
  - pas dans la définition

```
t = f(...)  # utilisation
```

```
def f(x, y): # définition
  ...
```

- Répondre à la question :
  - Comment faire correspondre un argument effectif à un argument formel ?

- Deux réponses possibles en Python :
  - en se basant sur la position dans la liste d'appel : argument positionnel
  - en se basant sur le nom de l'argument formel : argument nommé

```
t = f(a, b)
```

```
t = f(x=a, y=b)
```

Arguments positionnels

- Même position dans la définition et dans l'appel
- Souvent baptisés *args*

```
def f(x, y, z):
    return(math.sqrt(x*x+y*y+z*z))
```

```
print (f(a, b, c))
```

Arguments nommés

- Arguments nommés
- *Keyword Args*
- souvent baptisés *kwargs*

- Position non signifiante

```
def f(x, y, z):
    return(math.sqrt(x*x+y*y+z*z))
…
print (f(x=a, y=b, z=c))
print (f(y=b, z=c, x=a))
```

Arguments positionnels et nommés

- On peut mélanger

mais

- les arguments positionnels doivent figurer dans l'appel ***avant*** les arguments nommés

- Sinon : SyntaxError !

Arguments positionnels en nombre variable

- *x signifie que x est un tuple

**args*

Appelés parfois : *var args* ou *star args*

```
def f(*x):
    t=0
    for u in x:
        t+=u*u
    return(math.sqrt(t))
…
print (f(a, b, c))
```

opérateur *

- *unpacking*
- déballage

```
>>> uneListe = ['a', 'b', 'c']
>>> prem, *reste = uneListe
>>> prem
'a'
>>> reste
['b', 'c']
```

Arguments nommés en nombre variable

- **d est un dictionnaire dans lequel on trouve les arguments

***kwargs*

```
def g(**d):
    x=d['x']
    y=d['y']
    z=d['z']
    return(math.sqrt(x*x+y*y+z*z))
…
print (g(x=a, z=b, y=c))
```

Valeurs par défaut

- Dans la définition de la fonction
- Indiquer la valeur à utiliser si l'argument effectif est manquant dans l'appel
- Pour les derniers arguments uniquement
- Recourir aux arguments nommés en cas d'ambiguïté

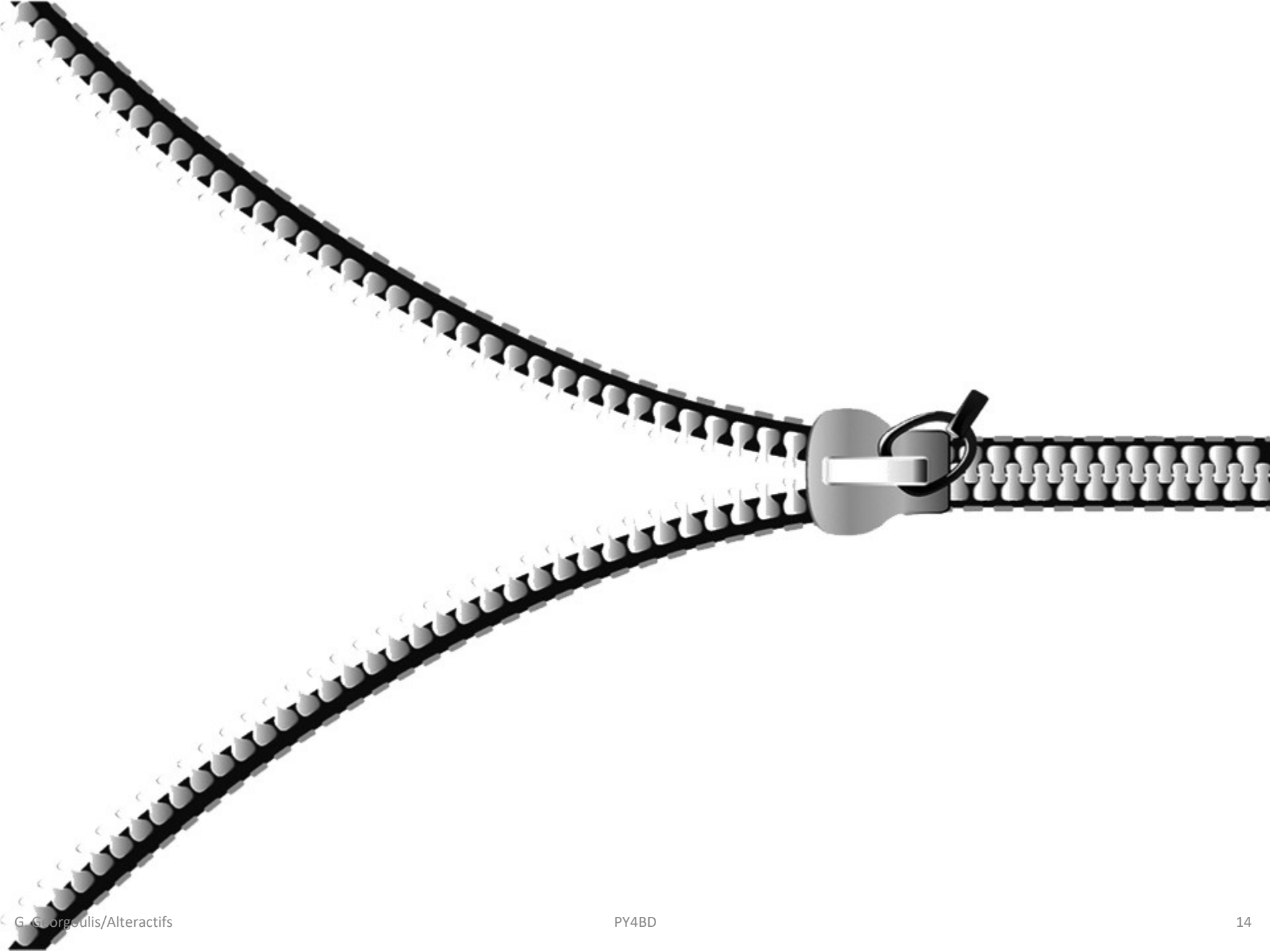
```
def f(x, y=0, z=0):
    return(math.sqrt(x*x+y*y+z*z))

print (f(a, b), f(a, z=2))
```

Défauts modifiables

- Les valeurs par défaut sont non modifiables
- plus de souplesse avec **None**
- l'algorithme de la fonction détermine une valeur par défaut calculée
  - on teste si x est l'objet **None**

```
def f(x = None):
    …
    if x is None:
        x = -b /(2*a)
    …
```



zip

- zip réunit deux objets itérables

```
a = [1, 2, 3]
neveux = ['Riri', 'Fifi', 'Loulou]
zipped = zip(a, neveux)
```

- dans cet exemple, un couple de listes est zippé en une liste de couples.

- Attention zip renvoit un objet zip (un itérateur)
- Le convertir en liste

```
zipped = list(zipped)
```

- Il n'y a pas de unzip. C'est zip qui est utilisé
- * est l'opérateur qui déballe les arguments (*splat*, *unpacking argument lists*)

```
l1, l2 = zip(*zipped)
```

Classeur Jupyter Zip.ipynb

time

- Mesurer le temps d'exécution

```
import time
start_time = time.time()
main()
print(
  f"{time.time() - start_time} seconds"
  )
```

Signature générique d'une fonction

- les arguments positionnels

avant

les arguments nommés

```
f(*args, **kwargs)
```

global

- La portée des noms (*scope*) peut être
  - globale
    - le fichier en cours
      - globals()
  - locale
    - la fonction en cours
      - locals()
- Priorités
  - local < global < prédéfini
- Une fonction ne peut pas modifier une variable globale à moins de l'avoir déclarée comme "global"

```
import math
# compter les appels à f
n=0

def f(x):
    global n
    #compter l'appel
    n+=1
    return x*math.sin(x)



def main():
   print(f(0), f(math.pi), f(5), sep="\t")
   print(n)

main()
```

Imbrication (*embedding),* emboîtement)

- la définition d'une fonction peut être locale à une autre fonction
  - Sa définition est incluse dans celle de la fonction qui l'utilise
  - Elle n'est pas définie en dehors
- Définir des fonctions auxiliaires sans polluer le scope global
  - Organisation du code
  - Eviter des effets de bords non désirés
  - Faciliter la maintenance
- Remarque : dans l'exemple ci-contre n reste défini au niveau global
  - Comment introduire n dans main ?
    - Essayer!!!

```
import math
# compter les appels à f
n=0
def main():

    def f(x):
        global n
        #compter l'appel
        n+=1
        return x*math.sin(x)



    print(f(0), f(math.pi), f(5),
          sep="\t")
    print(n)

main()
print(f(0)) # NameError : name 'f' is
              not defined
```

nonlocal

- Pour indiquer qu'un symbole est défini dans un niveau englobant
  - mais pas le niveau global

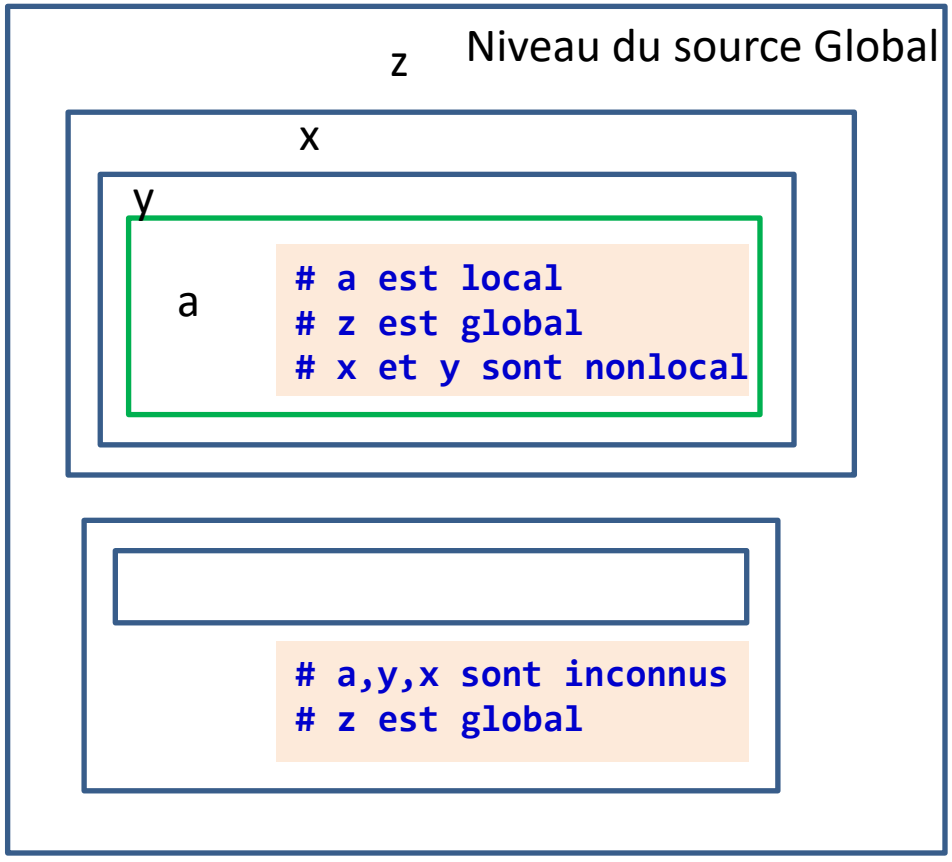
  on utilise nonlocal

```
import math
def main():
    # compter les appels à f
    n=0
    def f(x):
        nonlocal n
        #compter l'appel
        n+=1
        return x*math.sin(x)

    print(f(0), f(math.pi), f(5),
          sep="\t")
    print(n)

main()
```

Portée


z
Niveau du source Global
x
y
a
a est local
z est global
x et y sont nonlocal
a,y,x sont inconnus
z est global

Espaces de noms

- *namespace*
- gérés par des dictionnaires

```
builtins
```

- les fonctions et constantes prédéfinies par Python

```
__builtins__
dir(__builtins__)
```

- lister les noms locaux, globaux, connus en un point du code

```
locals()
globals()
```

- le préfixe du nom est le nom du module

```
math.pi
random.randint(a,b)
```

Module

- un module est un fichier .py qui contient des définitions
  - de fonctions
  - de variables

```
mon_module.py
```

- pour l'importer
  - même si on l'importe plusieurs fois dans un même projet, les variables ne sont initialisées que la 1ère fois

```
import mon_module
```

- pour en faire l'inventaire

```
dir(mon_module)
```

- si le module a été modifié et qu'il est nécessaire de le réimporter

```
from importlib import reload
reload(mon_module)
```

Package

- Un package est un ensemble de modules *.py
- En général, réunis dans un sous-dossier
- ***Jusqu'à la version 3.3***, il fallait faire figurer dans les sous-dossiers, un source spécial
  - __init__.py
    - souvent vide
    - permet de limiter les symboles qui seront exportés
    - tous les autres restent internes
- ***Depuis la version 3.3 (***PEP 420) Python dispose de packages implicites et ce fichier __init__.py n'est plus nécessaire. Il suffit de répartir les sources dans une hiérarchie de dossiers pour faire un package.
- Organiser le code

```
import mon_package #comme un module
```

```
all= ["ma_fonction"]
```

https://www.python.org/dev/peps/pep-0420/

Argument fonctionnel

- un argument à une fonction peut être une fonction
  - Noter l'usage de l'*underscore* dans le nom de la fonction
    - La fonction s'appelle map_ parce que Python connaît une fonction map qui fait presque la même chose.

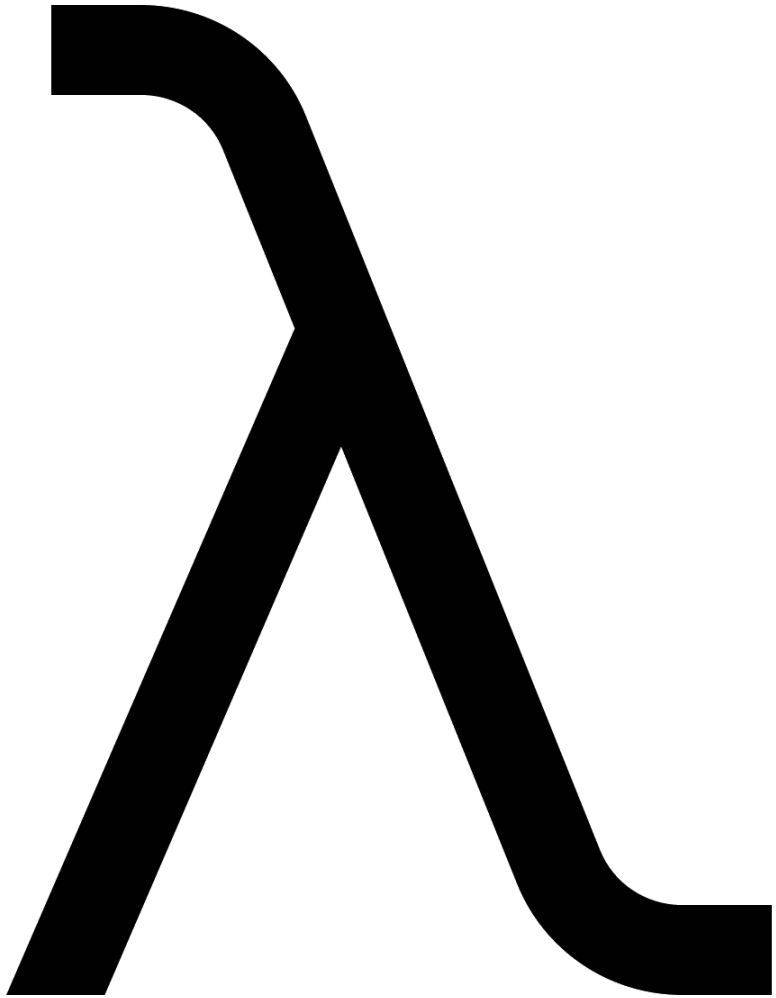
```
import math

def f(x):
    return x*x

def g(x):
    return math.sin(x)

def map_(func, l):
    t=[]
    for e in l:
        t.append(func(e))
    return t

print(map_(f, list(range(3))))
print(map_(g, list(range(3))))
```



lambda

λ ou en majuscule : Λ

- λ –calcul – thèse de A. Church (1930)
  - *"tout calcul se fait par une fonction"*

    Calculer une valeur
    Sans modifier les arguments
    Pas d'effet de bord

```
lambda x: x*x    # est une fonction
```

- Définir une fonction anonyme
  - absence du nom !
  - présence de la liste d'arguments
  - expression qui donne la valeur de retour

- en partie droite d'une affectation
  - def f(x):… est préféré

```
f = lambda x: x*x
print (f(12))
```

- Remplacer un argument fonctionnel

```
map(lambda x: x*x, list(range(3)))
```

Classeurs Jupyter Dir et Map

map- filter- reduce

- Map
  - appliquer une même fonction à tous les éléments d'une séquence
  - en construisant un itérateur sur une nouvelle séquence
    - Ex: transformer une liste de nombres en liste de str

```
list(map(lambda x: x**2, range(4)))
# [0, 1, 4, 9]
```

- Filter
  - appliquer une fonction de sélection à tous les éléments d'une séquence
  - l'itérateur obtenu donne les seuls éléments pour lesquels la fonction à retourné une valeur Vrai.

```
list(filter(lambda x: x%3==0, range(10)))
# [0, 3, 6, 9]
```

- Reduce
  - Combiner 2 à 2 les éléments d'une séquence en partant de la gauche

Classeurs Jupyter Map et Dir .ipynb


PARKER
GASOLINE GENERATOR
STARTER SW
ON
OFF
PILOT LAMP
VOLT METER
BREAKER
ON
OFF
PARKER
AC 115V/16A
VOLT SW
DC 12V 8.3A

yield

- Comme return

```
def g():
    …
    yield x
    suite
```

- g retourne alors un générateur

```
gen = g()
```

- chaque appel successif se fait par

```
next(gen)
```

- on récupère alors le résultat transmis par yield
- Contrairement à return, l'appel suivant reprend à l'instruction qui suivait le yield

Classeurs Jupyter Générateurs.ipynb

expression génératrice

- Liste de 10000 entiers en compréhension

```
[ i for i in range(10000) ]
```

- Expression génératrice

  Noter les parenthèses

  Les valeurs sont fabriquées lorsqu'il y en a besoin.

```
( i for i in range(10000) )
```

- Evaluation "paresseuse"

  lazy evaluation

Itérateur cf plus loin

Décorateur

- Signalé par @
- S'applique à la fonction dont la définition suit

```
@decorateur1
def func(x):
    …
```

- Un décorateur peut être une fonction
  - Un argument fonctionnel
  - Une seule valeur retournée, une fonction
  - dans l'écriture ci-contre g est fréquemment appelée "wrapper"

```
def decorateur1(f):
    def g (*args, **kwargs):
        …

    return g
```

- Dans d'autres langages : "annotation"

```
# si func a été décorée
func(x)
# équivaut à decorateur1(func)(x)
```

- Ne pas confondre avec le *design pattern* du décorateur

Décorateur simple

- Le décorateur le plus simple
  - pas de wrapper !

```
def decorateur1(f):
    # faire quelque chose ici
    print(f"Appel à {f.__name__}")
    return f
```

Si f est un objet de type "function",
essayer de faire dir(f) ou dir(type(f))

Empiler les décorateurs

Revient à composer les fonctions !

```
def decorateur1(f):
    ...

def decorateur2(f):
    ...

@decorateur1
@decorateur2
def myFunc(x):
   return ...


myFunc(0)

#fait la même chose que
décorateur1(
  decorateur2(
    myFunc))(0)
```

Décorateur à plusieurs arguments

- Un décorateur peut prendre des arguments en plus de l'argument fonctionnel
  - Le décorateur avec arguments est une fonction qui renvoie un décorateur sans argument

```
""" Décorateur répéter n fois """
def repeat(n):

    def repeater(f):
        def wrapper(*args, **kwargs):
            for i in range(n):
                f(*args,**kwargs)
        return wrapper

  return repeater

@repeat(20)
def bonjour():
    print("bonjour")


bonjour()
```

https://stackoverflow.com/questions/5929107/decorators-with-parameters

Classe qui décore une fonction

- Un décorateur peut être une classe
- Posséder la méthode __call__ (callable)

```
""" Decorateur class Repeat """

class Repeat:
    def __init__(self, n):
        self.count = n
    def __call__(self, f):
        def wrapper(*args, **kwargs):
            for i in range(self.count):
                f(*args,**kwargs)
        return wrapper

@Repeat(20)
def bonjour():
    print("bonjour")


bonjour()

# Repeat(20)(bonjour)()
```

Fonction qui décore une classe

- Une fonction qui prend pour argument une classe et retourne la classe décorée
- Par exemple pour instrumenter une méthode

```
def private(cls):
    cls.__getattribute__1 = cls.__getattribute__
    def getattribute2(self,name):
        if name =='personName':
            return cls.__getattribute__1(self,name)
        else:
            return len(str(cls.__getattribute__1(
                   self,name)))*"*"
    cls.__getattribute__ = getattribute2
    return cls



@private
class Person:
    def __init__(self, personName, phoneNum):
        self.personName = personName
        self.phoneNum = phoneNum

    def __str__(self):
        return f"{self.personName}, {self.phoneNum}"

p1 = Person( 'Paul', '0612345678')
p2 = Person( 'Marie-Louise', '0146789010')
print(p1)
print(p2)
#Paul, **********
#Marie-Louise, **********
```

Décorateurs prédéfinis

- @property
- @x.setter, getter, deleter

Définir un attribut avec les méthodes pour le modifier, le supprimer, l'obtenir

getter, setter, deleter sont appelés des mutateurs

Ce dispositif est justifié si les mutateurs font quelque chose de plus que = ou del

La propriété est "cachée" par __ et reste manipulable par les mutateurs prévus par le concepteur.

```
class UneClasse:
    def __init__(self,val):
        self.propriete = val

x = uneClasse(12)
print (x.propriete)
```

```
class UneClasse:
    def __init__(self,val):
        self.__propriete = val

    @property
    def propriete(self):
        return self.__propriete

    @propriete.setter
    def propriete(self, val):
        self.__propriete = val

x = uneClasse(12)
print (x.propriete)
```

functools

High order functions

fonctions de fonctions

- partial
- reduce
- @cache
- @wraps
- @singledispatch + register

et plus en https://docs.python.org/3/library/functools.html

functools.partial

- Fournir des arguments par défaut à une fonction existante

```
from functools import partial
```

- pow prend deux arguments
  - pow(num, exp)
- map demande ici une fonction à 1 argument
- partial crée une fonction à un seul argument à partir de pow

```
map(partial(pow, exp=3), range(10))
```

functools.reduce

- Complémentaire de map dans les traitements parallèles *map-reduce*

```
from functools import reduce
```

- Combiner 2 à 2 les éléments d'une séquence en partant de la gauche

```
# sum-like
reduce(lambda x,y: x+y, iterable)
```

- Possibilité de spécifier la valeur initiale

Classeurs Jupyter Map et Dir .ipynb

@functools.cache

- Pour améliorer le comportement des fonctions récursives
  - *memoize*
  - *last_recently_used cache*

```
from functools import cache
```

- Se souvenir des valeurs récemment calculées plutôt que de recommencer le même calcul

```
@cache
def fac(n):
   return n*fac(n-1) if n else 1
```

@functools.wraps

- Pour préserver
  - le nom de la fonction
    f.__name__
  - son help
    f.__doc__
  - l'introspection en général

- Recopier les attributs de la fonction décorée

```
from functools import wraps
```

```
def decorateur1(f):
    @wraps(f)
    def g (*args, **kwargs):
        …

    return g
```

@functools.singledispatch

- Faire une fonction générique
  - qui s'adapte au type du 1er argument qu'elle reçoit

```
from functools import singledispatch
```

generic function

```
@singledispatch
def f(x):
   print(x)
```

- Surcharger la fonction générique selon le type du premier argument

```
@f.register
def _(x:int):
   print("Entier", x)
```

```
@f.register
def _(x:list):
   print("Liste")
   for i,e in enumerate(x):
     print(i, e, sep=":")
```

itertools

Iterator building blocks

- Style fonctionnel
- Traitements itératifs sur des grands volumes de données

- chain
- dropwhile/takewhile
- groupby

et plus en https://docs.python.org/3/library/itertools.html

itertools.chain

- Construire un itérateur qui met à la suite plusieurs itérateurs

```
from itertools import chain
```

  - Les chiffres de 0 à 9
  - Les lettres de A à F
  - Les chiffres hexa

```
a = (chr(48+i) for i in range(6))
b = (chr(65+i) for i in range(6))
list(chain(a,b))
```

Classeur Jupyter itertoolschain

itertools. takewhile/dropwhile

- Construire un itérateur
  - qui filtre les valeurs d'un itérateur
  - par un prédicat

```
from itertools import dropwhile,
                      takewhile
```

- takewhile inclut les valeurs tant que le prédicat est vérifié

```
list(takewhile(lambda x: x%13,
               range(8,20)))
# [8, 9, 10, 11, 12]
```

- dropwhile exclut les valeurs jusqu'à ce que le prédicat soit vérifié

```
list(dropwhile(lambda x: x%13,
               range(8,20)))
# [13, 14, 15, 16, 17, 18, 19]
```

Classeur Jupyter takedropwhile

itertools.groupby

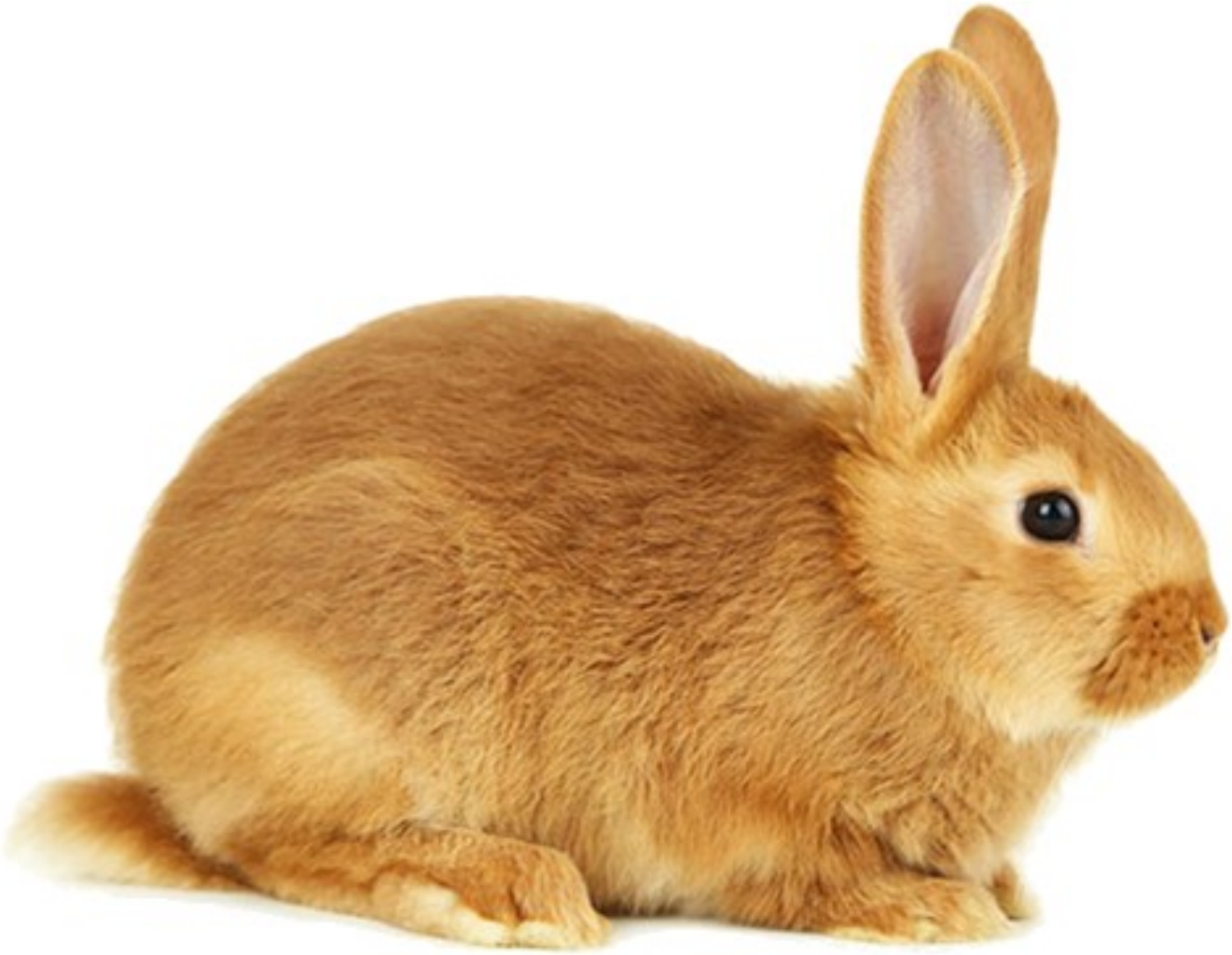
- Eléments supposés dans l'ordre des clés

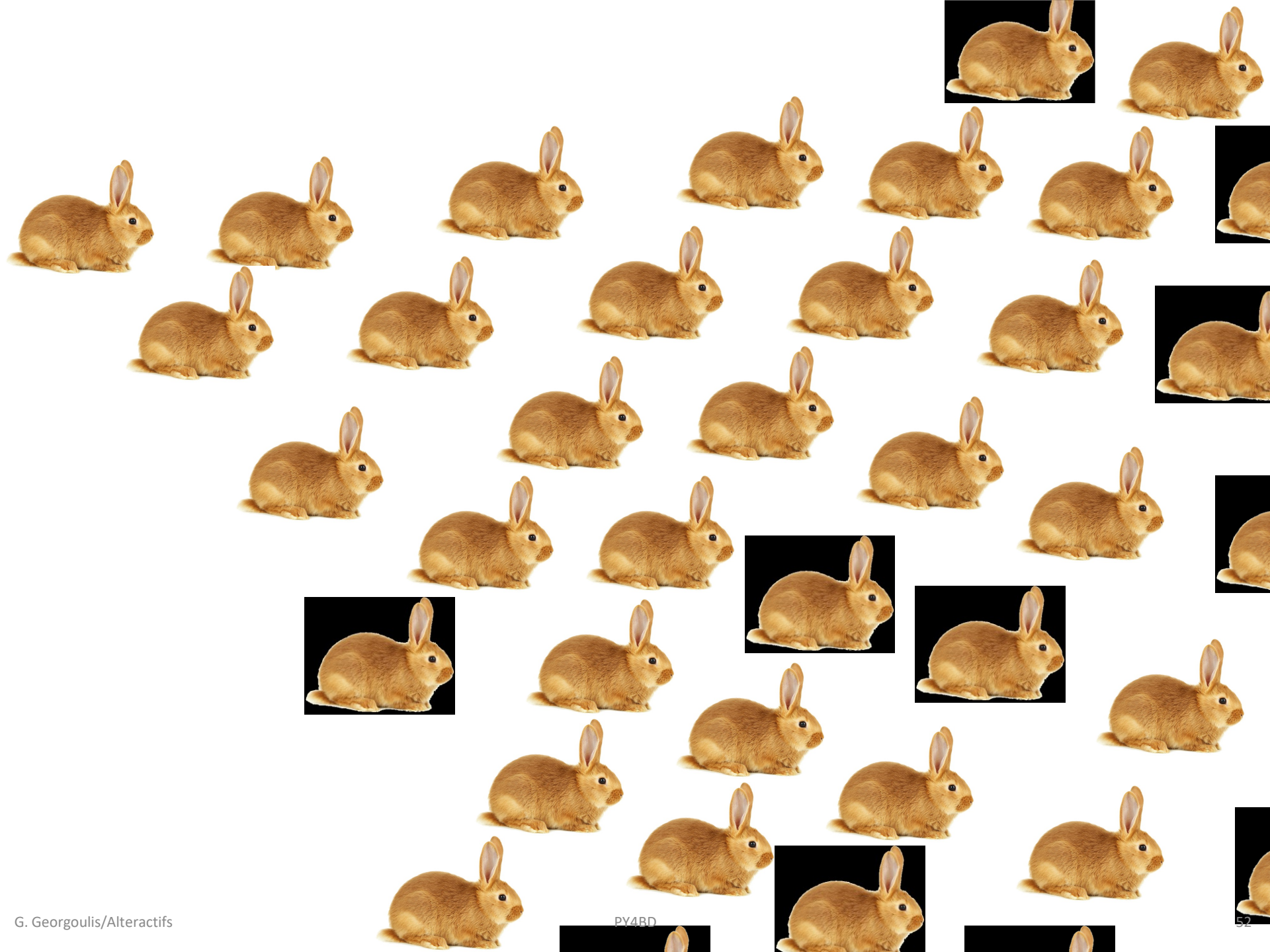
```
from itertools import groupby
```

- groupby produit un itérateur qui retourne
  - une clé k et pour chaque clé
  - son groupe
    - un itérateur qui liste les éléments de même clé

```
tu = 'aa', 'ab', 'ac', 'ba', 'bb'
for k, g in \
   groupby(tu, lambda x: x[0]):
   print(k, list(g))
# a ['aa', 'ab', 'ac']
# b ['ba', 'bb']
```

Classeur Jupyter ecgroupby





Exercice Fibonacci

- Nombres de Fibonacci
- 1, 1, 2, 3, 5, …
- le suivant est la somme des 2 précédents
- Modèle de l'évolution d'une population de lapins

- Lister les 10 premiers nombres de Fibonacci à l'aide
  - algorithme récursif fib_r(n)
  - algorithme itératif fib_i(n)
  - générateur fib_g()
- Instrumenter vos algorithmes pour compter les opérations effectuées

Classeur Jupyter Fibonacci.ipynb

Exercice Décorateur

- Ecrire un décorateur show_call qui présente la liste d'appel d'une fonction
- Ce genre de fonction est utile au debugging
- Reprendre l'un des exercices précédents pour des essais

```
@show_call
def func(x=0, y=0, Epsilon=1E-7):
    pass
```

```
func (1, Epsilon=12)
```

```
1, Epsilon=12
```

Les mots-clés ajoutés

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

Exercice Tortue

- Seymour Papert et al
- Langage Logo (1967)
- Sciences de l'éducation
- Approche de l'algorithmique

- Introduit une Tortue qui reçoit des instructions simples pour dessiner sur l'écran.

- Accessible dans Python
- Documenté en Turtle graphics

```
from turtle import *

showturtle()
hideturtle()

shape('turtle')

forward(100)
left(90)
right(90)
backward(100)

penup()
pendown()
```

Tortue, pas à pas(1/2)

- Dessiner un carré de côté 100

Changer la taille du carré :

- Dessiner un carré de côté 10

Puis

- Dessiner un carré de côté 25

Puis

- Dessiner un carré de côté 50

Pour chacun de ces énoncés, se poser la question : comment éviter d'écrire plusieurs fois la même chose (les mêmes instructions) ?

Tortue, pas à pas (2/2)

- Dessiner un triangle équilatéral de côté 100
- Dessiner un hexagone régulier de côté 100
- Dessiner un pentagone régulier de côté 100
- Dessiner un heptagone, un octogone, un enéagone, un décagone... (réguliers)
- Un polygone régulier à n côtés

Merci !

- Restons en contact :
  - Georges Georgoulis – ggeorgoulis@alteractifs.org – 06 12 68 40 06

  - Alteractifs

    Coopérative d'activité et d'entrepreneurs www.alteractifs.org