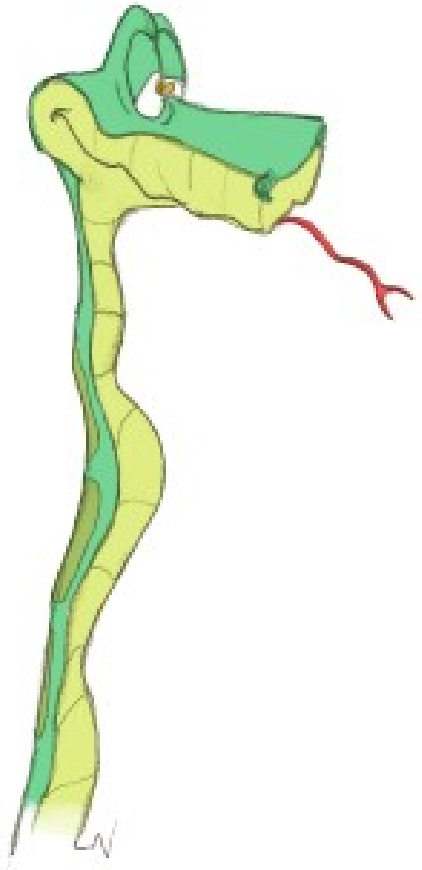
Notes de cours Python

Bob CORDEAU
bob@kordeo.eu

Sommaire



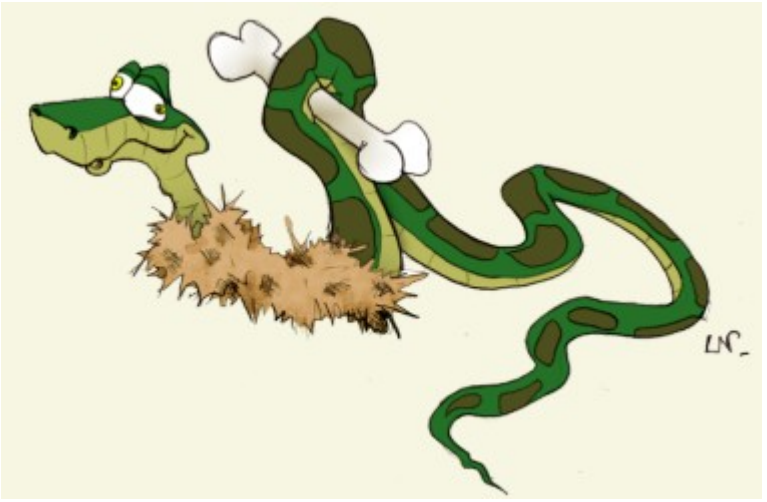
GvR


MONTY
PYTHON

Chapitre 1/11

Introduction

- Pourquoi choisir Python ?
- Environnement matériel
- Environnement logiciel
- Langages
- Production des programmes
- Méthodologie de programmation
- Historique



Pourquoi choisir Python ?

- Python n'est pas le meilleur langage pour tout. Pour travailler sur du texte, Perl est meilleur. Pour de l'intelligence artificielle, Lisp est meilleur. Pour de l'embarqué, C est meilleur. Pour de l'asynchrone, JavaScript est meilleur.
- Toutefois, il est très bon pour tout. Python est excellent pour écrire des scripts rapides, pour analyser du texte, pour faire un site web, de l'administration système, des interfaces graphiques, du calcul scientifique, de la 3D, des robots d'indexation, etc.
- Python s'interface nativement avec le code de nombreux autres langages (*a glue language*)

Caractéristiques principales

- Python est **portable** (mais… Python 2 / Python 3)
- la syntaxe de Python est très simple, compacte et **très lisible**
- Python gère automatiquement ses ressources (mémoire, descripteurs de fichiers…)
- Il supporte plusieurs styles de programmation
- Il est orienté objet, supporte l'héritage multiple et la surcharge des opérateurs
- …
- Et de plus, Python est gratuit !

Plan

- Introduction
  - Pourquoi choisir Python ?
  - Environnement matériel
  - Environnement logiciel
  - Langages
  - Production des programmes
  - Méthodologie de programmation
  - Historique

L'ordinateur

Automate déterministe à composants électroniques

Comprend au moins :

- Microprocesseur : UC (Unité de Contrôle), UAL (Unité arithmétique et Logique), horloge, mémoire cache rapide…
- RAM (instructions et données) organisée en octets
- Périphériques : entrées/sorties, mémoires de masse (disques durs, CD-ROM, clés USB, réseau, disquettes...)

Plan

- Introduction
  - Pourquoi choisir Python ?
  - Environnement matériel
  - Environnement logiciel
  - Langages
  - Production des programmes
  - Méthodologie de programmation
  - Historique

Deux sortes de programmes

- Le système d'exploitation : gestion des ressources matérielles et logicielles. Jadis textuel, maintenant graphique. Souvent multitâche et parfois multiutilisateur

- Les programmes applicatifs dédiés à des tâches particulières

- Dans tous les cas, un programme contient une série de commandes contenues dans un programme *source* écrit dans un langage « compris » par l'ordinateur

Plan

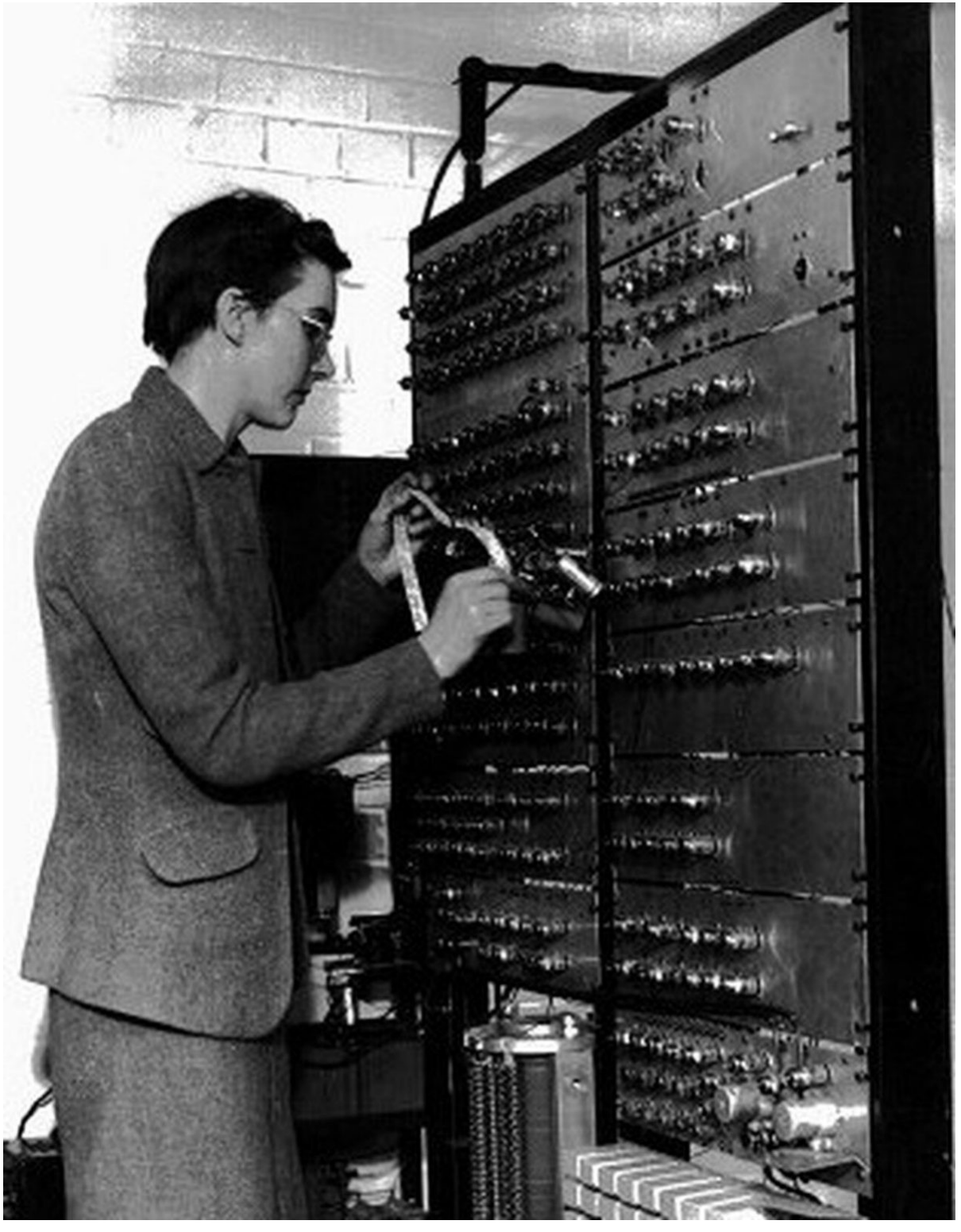
- Introduction
  - Pourquoi choisir Python ?
  - Environnement matériel
  - Environnement logiciel
  - Langages
  - Production des programmes
  - Méthodologie de programmation
  - Historique

Des langages de différents niveaux

- Tout processeur possède un langage propre exécutable formé de 0 et de 1 : le langage machine. Il n'est pas portable et c'est le seul que l'ordinateur « comprend »

- Le langage d'assemblage, codage mnémonique non portable du langage machine. Il est traduit en langage machine par un *assembleur* (Kathleen Booth)

- Les langages de hauts niveaux, permettent le portage entre machines. Traduits en langage machine par un *compilateur* ou *interpréteur*



Plan

- Introduction
  - Pourquoi choisir Python ?
  - Environnement matériel
  - Environnement logiciel
  - Langages
  - Production des programmes
  - Méthodologie de programmation
  - Historique

Deux techniques de production

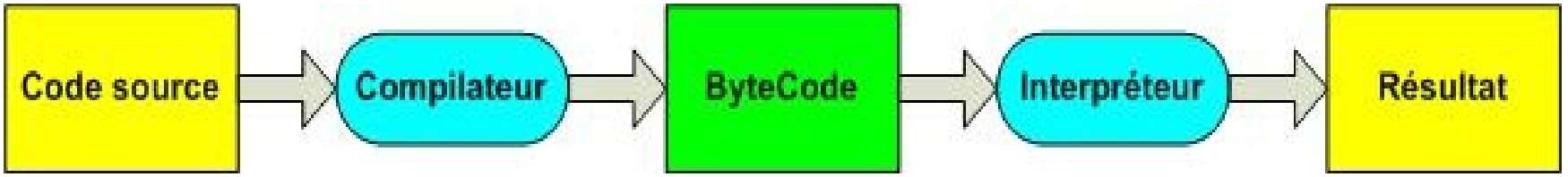
- Compilation : traduction du source en langage objet : *trois phases d'analyse*, une de *production du code objet* plus celle d'*édition de liens*. Technique très contraignante mais offrant des codes rapides

- Interprétation : chaque ligne du source est traduite en instructions exécutables. Technique souple, mais codes peu performants. Aucun code objet généré : l'interpréteur doit être utilisé à *chaque* exécution

Technique de production de Python

- Technique mixte : interprétation du byteCode compilé.
  Bon compromis entre facilité de développement et rapidité d'exécution

- Le byteCode (forme intermédiaire) est portable sur tout ordinateur muni de la machine virtuelle Python



Plan

- Introduction
  - Pourquoi choisir Python ?
  - Environnement matériel
  - Environnement logiciel
  - Langages
  - Production des programmes
  - Méthodologie de programmation
  - Historique

Le génie logiciel

Plusieurs méthodologies possibles :

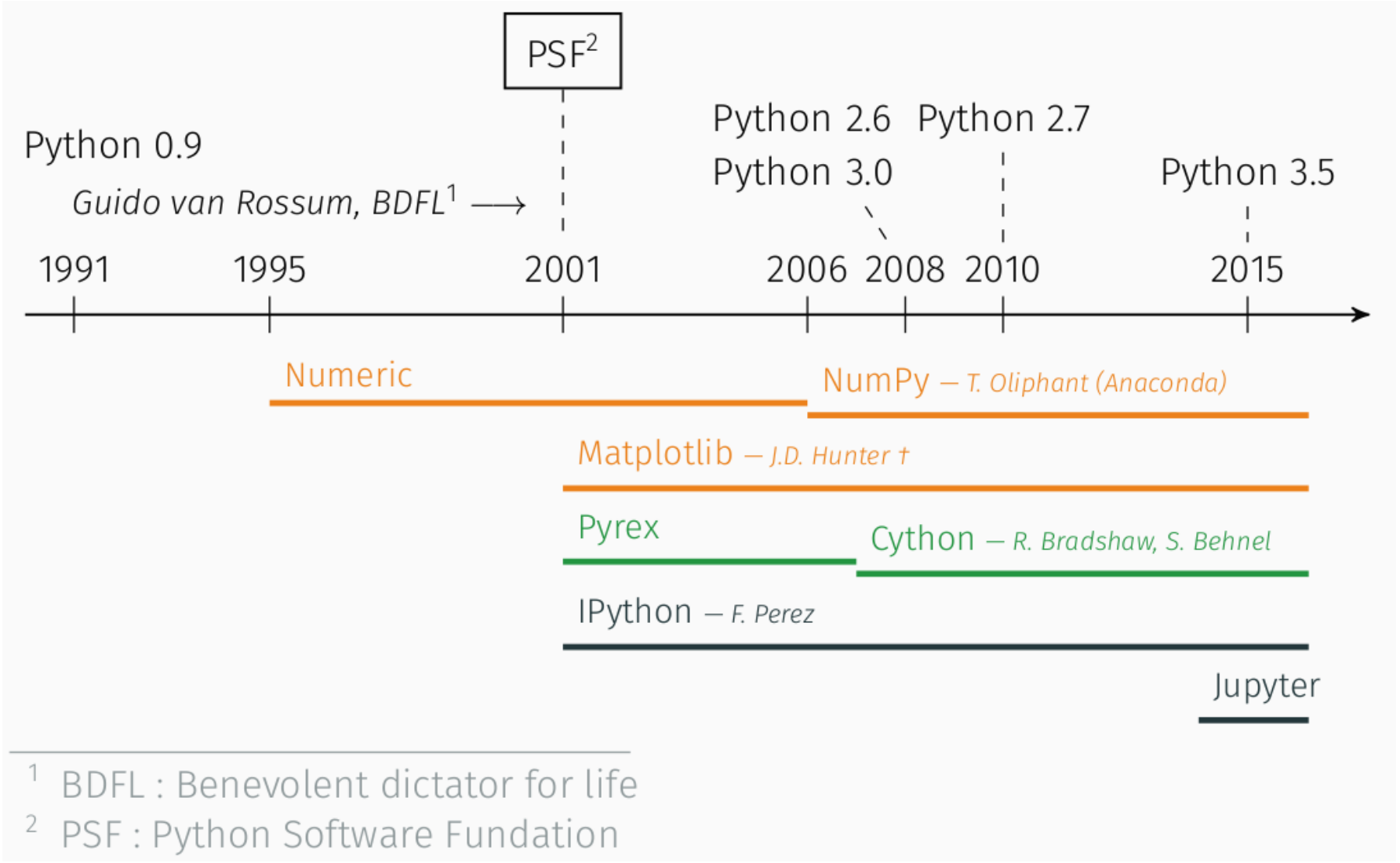
- Procédurale : analyse descendante et remontante par raffinements successifs : décomposition d'un problème complexe en *sous-programmes* plus simples. Ce modèle structure d'abord les actions
- Objet : on conçoit des fabriques (*classes*) qui servent à produire des composants (*objets*) qui contiennent des données (*attributs*) et des actions (*méthodes*). Les classes dérivent (*héritage et polymormisme*) de classes de base

Python offre les deux techniques (paradigmes).

Plan

- Introduction
  - Pourquoi choisir Python ?
  - Environnement matériel
  - Environnement logiciel
  - Langages
  - Production des programmes
  - Méthodologie de programmation
  - Historique

Bref historique



[1] BDFL : Benevolent dictator for life
[2] PSF : Python Software Fundation

Interlude


SPAM
GOLDEN HONEY GRAIL
Wicked-awesome SPAMALOT™ Collector's edition
NET WT 12 OZ (340g)
Hormel Foods

L'interpréteur de base : `python` (on lui préfère `ipython`)

```
Python 3.9.7 (default, Sep 16 2021, 13:09:58)
[GCC 7.5.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>> 2 + 3                    # Addition d'entiers
5
>>> 1 + 0.03                 # Entier et flottant
1.03
>>> (2 + 3) / 7              # Division flottante
0.7142857142857143
>>> 12 // 5                  # Division entière
2
>>> 0.1 + 0.2                # Le classique !
0.30000000000000004
>>> msg = "73, F6KRK !""     # Affectation d'un message
>>> print(msg)               # Affichage d'une valeur
73, F6KRK !
```

- Installer Python
  - Le site officiel `www.python.org`
  - Anaconda `www.anaconda.com/products/distribution`
  - Installation recommandée, `Miniconda3` :
  - À partir du site : `https://perso.limsi.fr/pointal/`

    Installation *via* miniconda 3 :

    `python:installation:accueil#installation_de_python_via_miniconda`

    Gestion *via* conda :

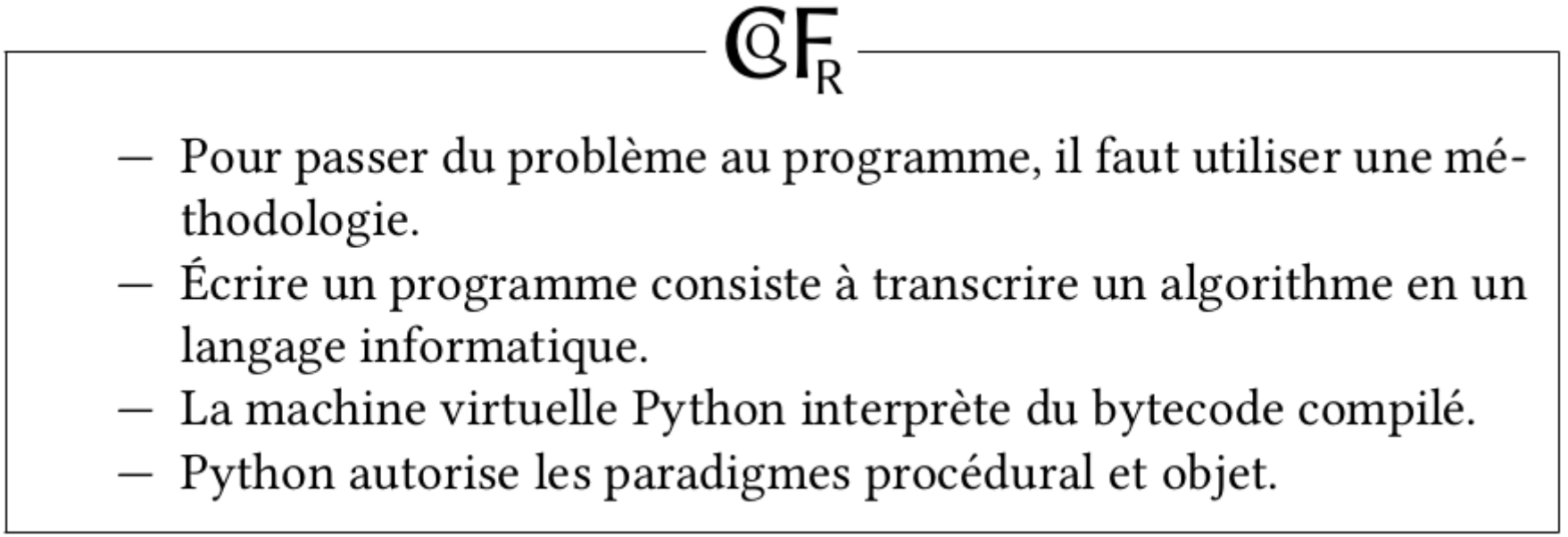
    `perso.limsi.fr/pointal/python:conda:accueil`
- Environnement intégré : `anaconda-navigator`
  - `ipython`
  - `Spyder`
  - `Jupyter notebook`

Ressources

- Documentation officielle `docs.python.org/3/`
- Archive `blog.f6krk.org/formation-au-langage-python/`
- Liens Python `perso.limsi.fr/pointal/liens:python_links`
- Poly Python `perso.limsi.fr/pointal/python:courspython3`
- Memento `perso.limsi.fr/pointal/python:memento`
- Turtle `perso.limsi.fr/pointal/python:turtle:accueil`
- …

Mieux utiliser `conda`, `ipython` & `spyder`

Programmer en Python

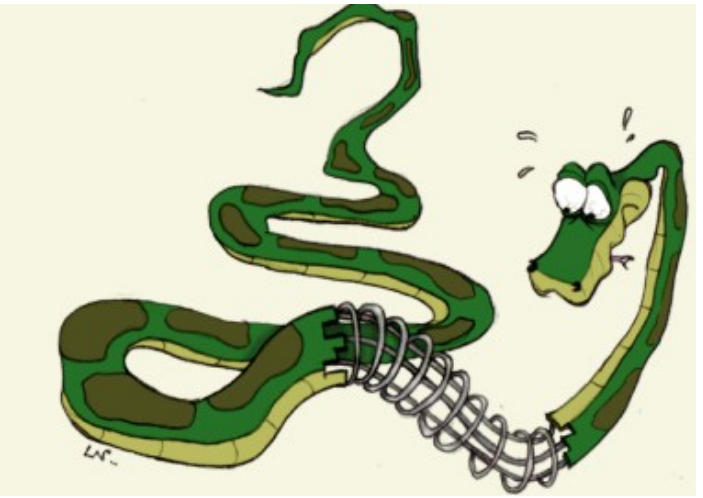


— Pour passer du problème au programme, il faut utiliser une méthodologie.
— Écrire un programme consiste à transcrire un algorithme en un langage informatique.
— La machine virtuelle Python interprète du bytecode compilé.
— Python autorise les paradigmes procédural et objet.

Chapitre 2/11

La « calculatrice » Python

- Identificateurs et mots-clés
- Les types simples
- Données, variables et affectation
- Les entrées-sorties



Identificateurs

- Un identificateur Python est une suite non vide de caractères, de longueur quelconque, formée d'un caractère de début (lettre Unicode, caractère souligné) et, éventuellement, d'un ou plusieurs caractères de continuation (lettre Unicode, caractère souligné ou chiffre)

- Les identificateurs sont sensibles à la casse (distinction minuscule/majuscule) et ne doivent pas faire partie des mots réservés de Python 3

Mots-clés (Python 3.9+)

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

Plan

La « calculatrice » Python

- Identificateurs et mots-clés
- Les types simples
- Données, variables et affectation
- Les entrées-sorties

Types de données

En programmation informatique, un type de donnée, ou simplement un **type**, définit la nature des valeurs que peut prendre une donnée, ainsi que les opérateurs qui peuvent lui être appliqués.

La fonction native `type()` donne le type de l'objet :

```
>>> type('Bob')
<class 'str'>
>>> type(69)
<class 'int'>
>>> type([1, 2, 3])
<class 'list'>
```

Chaînes de caractères `str`

Trois notations disponibles :

1. Les guillemets permettent d'inclure des apostrophes :

```
>>> c1 = "l'eau vive"
```

2. Les apostrophes permettent d'inclure des guillemets :

```
>>> c2 = ' est "froide"'
```

3. Les triples guillemets (ou triples apostrophes), notation multiligne qui conservent la mise en forme :

```
>>> c3 = """
Usage :
    -h : help
    """
```

Encodage des chaînes

- L'encodage historique ASCII (Python 2) est devenu insuffisant
- Python 3 encode le type `str` en **Unicode**
- Trois autres encodages sont utiles :
  - le type **immutable** `bytes` de données binaires
  - le type **mutable** `bytearray` de données binaires
  - le type `raw` qui n'interprète pas les caractères spéciaux (très utile dans les expressions régulières)

Strings : opérations de base

Longueur :

```
>>> len("F6KRK")
5
```

Répétition :

```
c4 = "Fi !"
>>> c5 = c4 * 3
'Fi !Fi !Fi !'
```

Concaténation :

```
>>> c1 = "abc"
>>> c2 = "defg"
>>> c3 = c1 + c2
>>> 'abcdefg'
```

Méthodes sur les chaînes

Strings : indiçage et extraction

L'indiçage commence à `0` :

```
>>> c = 'Fi !Fi !Fi !'
>>> c[3]
'!'
>>> c[0]
'F'
```

Découpage (*slicing*) :

```
>>> c[1:3]
'i !'
>>> c[4:]
'Fi !Fi !'
```

Attention : Les strings ne sont pas modifiables !

Strings : formatage « à la C » (le formatage historique)

```
>>> n, pi = 100, 3.1415926535897931
>>> print("%d en base 10 : %d" % (n, n))
'100 en base 10 : 100'
>>> print("%d en base 8 : %o" % (n, n))
'100 en base 8 : 144'
>>> print("%d en base 16 : %x" % (n, n))
'100 en base 16 : 64'
>>> print("%4.2f" %(pi))
3.14
```

Strings : formatage « à la C »

```
>>> print("%.5e" %(pi))
3.14159e+000
>>> print("%g" %(pi))
3.14159
>>> msg = "Résultats sur %d échantillons : %4.2f" % (n,
  pi)
>>> print(msg)
'Résultats sur 100 échantillons : 3.14'
```

formatage avec `format()`

- Une chaîne possède une méthode `format()` utilisant des paramètres positionnels :

```
In [1]: var = 5

In [2]: print("Ma variable vaut {} et sa moitié vaut {}".format(var, var/2))
Ma variable vaut 5 et sa moitié vaut 2.5
```

- Cette méthode permet aussi de préciser l'ordre d'affichage des éléments en indiquant cet ordre dans les accolades (le premier élément formaté commençant à 0) :

```
In [3]: print("J'ai obtenu la valeur {1} en calculant {0}*{0}".format(var, var**2))
J'ai obtenu la valeur 25 en calculant 5*5
```

f-strings – bases

C’est la méthode utilisée depuis Python 3.6

```
>>> x = 4
>>> titre = "Python 3"
>>> print(f"J’ai commandé {x} exemplaires de {titre}.")
'J’ai commandé 4 exemplaires de Python 3.'
>>> print(f"J’ai commandé {x * 100} exemplaires de
  {titre.upper()}.")
'J’ai commandé 400 exemplaires de PYTHON 3.'
>>> print(f"Bonjour {'Guido'.upper()}.")
'Bonjour GUIDO'
```

f-strings : directives de formatage

```
>>> f"Ils sont {x:5}"           # entiers alignés à droite
'Ils sont    4'
>>> f"Le titre est {titre:12}" # chaînes alignées à gauche
'Le titre est Python 3   '
>>> f"Ils sont {x:>12}"  # Chaîne alignée à droite
'Ils sont           4'
>>> f"Ils sont {x:<12}"  # Alignée à gauche
'Ils sont 4          '
>>> f"Ils sont {x:-^12}" # Centrée avec un caractère de
  remplissage
'Ils sont -----4------'
```

Les booléens `bool`

- Deux valeurs : `False, True`
- Opérateurs de comparaison : `==` , `!=`, `>`, `>=`, `<`, `<=`

```
>>> 2 > 8
False
>>> 2 <= 8 < 15
True
```

- Opérateurs logiques : `not`, `or`, `and` (principe du shortcut)

```
>>> (3 == 3) or (9 > 24)
True
>>> (9 > 24) and (3 == 3)
False
```

Les entiers `int`

Opérations de base :

```
>>> 20 + 3
23
>>> 20 - 3
17
>>> 20 * 3
60
>>> 20 ** 3
8000
>>> 20 // 3 # division entière
6
>>> 20 % 3      # modulo
2
```

Taille limitée que par la mémoire de la machine :

```
>>> 1_295 ** 34  # _ lisibilité
6564053713044849619288097477263
  6180455628807147888171462166 0
  8634082326887740806420915760 0
  9988784790039062 5
```

Les bases :

```
07 + 01    # 8
oct(7+1)    # '010' (octal)
0x9 + 0x2    # 11
hex(9+2)    # '0xb' (hexa)
```

Les flottants `float`

- Indiqués par un point décimal ou une notation exponentielle :

```
>>> 2.718
>>> 25e-4
>>> .314
>>> 2_347_518.23   # _ améliore la lisibilité
```

- Le module « math » fournit les constantes et opérations usuelles :

```
>>> import math
>>> print(math.pi)
3.141592653589793
```

Les complexes `complex`

Notation cartésienne formée de deux flottants.

La partie imaginaire est suffixée par `j` :

```
>>> 1j
1j
>>> (2 + 3j) + (4 - 7j)
(6-4j)
>>> print((9 - 5j).real, (9 - 5j).imag)
9.0 -5.0
>>> abs(3 + 4j)          # module
5
```

Plan

La « calculatrice » Python

- Identificateurs et mots-clés
- Les types simples
- Données, variables et affectation
- Les entrées-sorties

Données, variables et affectation

- On nomme (on *référence*) une donnée par une *variable*
- On affecte une variable par une valeur grâce au signe = (*différent de l'égalité en math !*) :

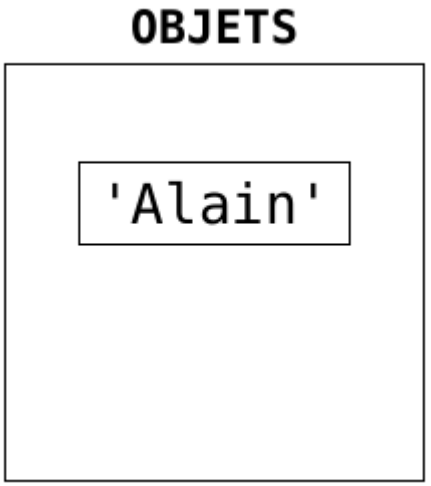
```
>>> a = 2                    # en fait : a « reçoit » 2
>>> uneVariable = 5
```

- Une variable peut changer de valeur ou de type :

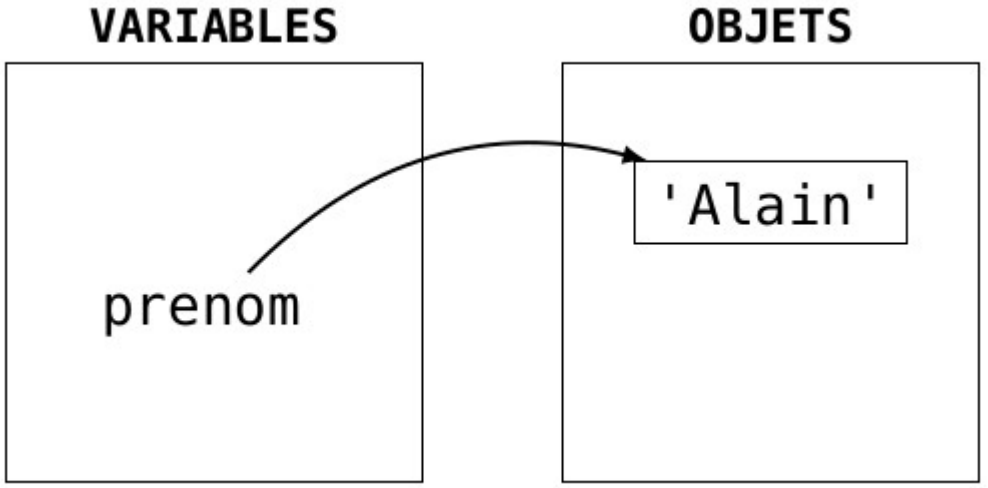
```
>>> uneVariable = uneVariable + a
>>> a = a + 1                # 3 (incrémentation)
>>> a = a - 1                # 2 (décrémentation)
>>> a = "un nouveau type" # str
>>> b = str(17)              # '17' (transtypage ou ‘cast’)
```

Représentation graphique de l'affectation

Espace des objets :



Affectation :



Plan

La « calculatrice » Python

- Identificateurs et mots-clés
- Les types simples
- Données, variables et affectation
- Les entrées-sorties

Les entrées-sorties

- Entrée ou saisie : **`input()`**

```
>>> # affiche une invite (« prompt ») et attend une saisie
  textuelle :
>>> n = input('Entrez un entier : ')
```

- Sortie ou affichage : **`print()`**

```
>>> a = 2
>>> b = 5
>>> print(a, b)
2 5
>>> print("Somme de deux entiers ", a + b)
7
```

Calculatrice Python – exercices

La calculatrice Python

- On peut exécuter des instructions Python directement dans un interpréteur, ou bien stocker des scripts à l'aide d'un éditeur, ou mieux, d'un IDE.
- Python 3.8 réserve 35 mots-clés.
- Le respect d'une politique cohérente de nommage et de commentaires améliore la lisibilité des sources.
- Les variables référencent les objets.
- Les objets possèdent un typage fort et les variables un typage dynamique.
- Les chaînes de caractères sont indexables et immutables. Pour les formater, on privilégie la syntaxe des *f-string*.
- Python permet de trouver rapidement une documentation.

Chapitre 3/11

Le contrôle du flux d'instructions

- Indentation significative et instructions composées
- La sélection
- La boucle **`while`**
- La boucle **`for`**
- Les ruptures de séquences
- Traitement des erreurs : les exceptions

Indentation significative
Instructions composées

- Python utilise la notion d'**indentation significative**. Cette syntaxe, légère et visuelle, met en lumière un bloc d'instructions
- Une **instruction composée** comprend :
  - une ligne d'introduction terminée par le caractère « deux-points » (:)
  - suivie d'un bloc d'instructions indenté. On utilise par convention quatre espaces par indentation. On n'utilise pas les tabulations mais uniquement les espaces
- On peut « imbriquer » des instructions composées pour réaliser des structures de décision complexes

Choisir : `if - [elif] - [else]`

```
>>> x = 5
>>> if x < 0:
...     print("x est négatif")
... elif x % 2 != 0:
...     print("x est positif et impair")
...     print ("ce qui est bien aussi !")
... else:
...     print("x n'est pas négatif et est pair")
...
```

Syntaxe compacte d'une alternative

```
>>> x = 4
>>> y = 3
>>> if x < y:                                    # Écriture classique
...     plus_petit = x
... else:
...     plus_petit = y
...
>>> print("Plus petit :", plus_petit)
Plus petit : 3
>>> plus_petit = x if x < y else y               # Utilisation de l'opérateur ternaire
>>> print("Plus petit :", plus_petit)
Plus petit : 3
>>> print(f"Plus petit : {x if x < y else y}")  # Affichage formaté
Plus petit : 3
```

Exemple d'instruction composée imbriquée

```
t = float(input("Température (°C) ? "))
print("Température 't' en degrés Celsius")
if t <= 0:
    print('Négative ou nulle : risque de gel')
else:
    print('Positive')
    if t > 25:
        print('Plus de 25 °C')
        print('Prévoir tee-shirt ou veste légère')
    elif t > 18 :
        print('Douce mais sans plus')
    else
        print('Positive mais <= 18 °C. Sortez couverts')
print("Évaluation terminée")
```

Plan

Le contrôle du flux d'instructions

- Indentation significative et instructions composées
- La sélection **`if`**
- La boucle **`while`**
- La boucle **`for`**
- Les ruptures de séquences
- Traitement des erreurs : les exceptions

Répéter sous conditions : boucle `while`

```
>>> x, cpt = 257, 0
>>> sauve = x
>>> while x > 1:
...     x = x // 2      # Division avec troncature
...     cpt = cpt + 1   # Incrémentation
...
>>> print("Approximation de log2 de", sauve, ":", cpt)
Approximation de log2 de 257 : 8
```

Saisie filtrée

```
n = int(input('Entrez un entier [1 .. 10] : '))
while not(1 <= n <= 10):
    n = int(input('Entrez un entier [1 .. 10], S.V.P. : '))
```

Si la saisie est compliquée, on peut utiliser une variable booléenne :

```
saisie_ok = False
while not saisie_ok:    # Tant que saisie_ok est False, faire :
    a = int(input("Entrez un nombre entier de 1 à 100 divisible par 3 : "))
    b = int(input("Entrez un nombre entier pair supérieur à " + str(a) + ": "))
    saisie_ok = (1 <= a <=100) and (a % 3 == 0) and (b % 2 == 0) and (a < b)
print("Ok")
```

Plan

Le contrôle du flux d'instructions

- Indentation significative et instructions composées
- La sélection **`if`**
- La boucle **`while`**
- La boucle **`for`**
- Les ruptures de séquences
- Traitement des erreurs : les exceptions

Parcourir : `for` (1/2)

```
>>> for lettre in "ciao":    # On peut itérer sur les caractères des chaînes
...     print(lettre, lettre.upper())
...
c C
i I
a A
o O
>>> for x in [2, 'a', 3.14]:  # Notation pour une liste de valeurs (cf. chap. 4)
...     # Le typage dynamique de Python permet à x de recevoir à
...     # chaque itération une valeur d'un type différent
...     print(x, x*2)
...
2 4
a aa
3.14 6.28
```

Parcourir : `for` (2/2)

Un itérable désigne un type de conteneur que l'on peut parcourir élément par élément :

```
>>> for i in range(6):          # Générateur de séquences d'entiers à la demande
...     print(i, i ** 2)
...
0 0
1 1
2 4
3 9
4 16
5 25
>>> nb_voyelles = 0
>>> for lettre in "Python est un langage fort sympa":
...     if lettre.lower() in "aeiouy":
...         nb_voyelles = nb_voyelles + 1
...
>>> nb_voyelles
10
```

Plan

Le contrôle du flux d'instructions

- Indentation significative et instructions composées
- La sélection **`if`**
- La boucle **`while`**
- La boucle **`for`**
- Les ruptures de flux en séquence
- Traitement des erreurs : les exceptions

Rupture de séquence (1/2)

Interrompre une boucle : `break`

```
for x in range(1, 11):
    if x == 5:
        break
    print(x, end=" ")  # end=" " remplace le retour

print("Boucle interrompue pour x =", x)
```

Ce qui produit :

```
1 2 3 4 Boucle interrompue pour x = 5
```

Rupture de séquence (2/2)

Court-circuiter une boucle : `continue`

```
for x in range(1, 11):
    if x == 5:
        continue
    print(x, end=" ")
```

La boucle a sauté la valeur 5 :

```
1 2 3 4 6 7 8 9 10
```

Plan

Le contrôle du flux d'instructions

- Indentation significative et instructions composées
- La sélection **`if`**
- La boucle **`while`**
- La boucle **`for`**
- Les ruptures de séquences
- Traitement des erreurs : les exceptions

Les exceptions

Dès que Python se trouve en échec, il lève une exception et la fait remonter jusqu'à ce qu'elle soit interceptée, soit dans le code, soit jusqu'à l'OS. Elle traverse toutes les couches de code comme une bulle d'air remonte à la surface de l'eau et s'affiche à l'interception :

```
In [1]: 1 / 0
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
/tmp/ipykernel_9691/1455669704.py in <module>
----> 1 1 / 0

ZeroDivisionError: division by zero
```

Exceptions : `try - except`

Gérer une exception permet d'intercepter une erreur pour éviter un arrêt du programme. On capte et on « gère » la bulle :

```
from math import sin

for x in range(-4, 5):  # -4, -3, -2, -1, 0, 1, 2, 3, 4
    try:
        print('{:.3f}'.format(sin(x)/x), end=" ")
    except ZeroDivisionError:   # Toujours fournir un type d'exception
        print(1.0, end=" ")     # Gère l'exception en 0
# -0.189 0.047 0.455 0.841 1.0 0.841 0.455 0.047 -0.189
```

Gérer ses propres exceptions

L'instruction **raise** permet de lever volontairement une exception :

```
if (x < 0) or (x > 1):
    raise ValueError, "la valeur n'est pas dans [0 .. 1]"
```

(voir la liste des exceptions standard dans la documentation intégrée)

Contrôle du flux d'instructions – exercices

Contrôle du flux d'instructions

@F$_R$

- Python est orienté « présentation » grâce aux indentations des instructions composées.
- Le flux séquentiel d'instructions est modifié par :
  - une instruction de choix (`if`),
  - une boucle de parcours d'un itérable (`for`),
  - une boucle conditionnelle (`while`),
  - une rupture de séquence (`break`, `continue`).
- Les exceptions gèrent efficacement les erreurs d'exécution.

Chapitre 4/11

Conteneurs standard

- Les séquences
- Retour sur les références
- Les dictionnaires
- Les ensembles

Conteneur & séquence

- **Conteneur** : objet composite destiné à contenir d'autres objets.
- **Séquence** : conteneur ordonné d'éléments indicés par un index (entier) indiquant leur position dans le conteneur. Les séquences sont numérotées à partir de 0.

- Python dispose de trois types prédéfinis de séquences : **chaîne**, **tuples** et **liste**

Les opérations sur les objets de type séquentiel

Les types prédéfinis de séquences Python (chaîne, liste et tuple) ont en commun les opérations résumées dans le tableau suivant, où `s` et `t` désignent deux séquences du même type, `x` un élément de la séquence et `i`, `j` et `k` des entiers :

Opération	Signification
`x in s`	`True` si `s` contient `x`, `False` sinon
`x not in s`	`True` si `s` ne contient pas `x`, `False` sinon
`s + t`	concaténation de `s` et `t`
`s * n` ou `n * s`	`n` copies (superficielles) concaténées de `s`
`s[i]`	i^e élément de `s` (à partir de 0)
`s[i:j]`	tranche de `s` de `i` (inclus) à `j` (exclu)
`s[i:j:k]`	idem avec un pas de `k`
`len(s)`	nombre d'éléments de `s`
`max(s)`, `min(s)`	plus grand, plus petit élément de `s`
`s.index(i)`	indice de la 1^{re} occurrence de `i` dans `s`
`s.count(i)`	nombre d'occurrences de `i` dans `s`

Les tuples (1/2)

Un tuple est une collection ordonnée immutable d'éléments éventuellement hétérogènes. Ses éléments sont séparés par des virgules, et optionnellement entourés de parenthèses.

```
>>> mon_tuple = ('a', 2, [1, 3])        # Tuple de trois éléments
>>> ton_tuple = 'un', 'deux', 'trois'   # Tuple de trois éléments
>>> s = (2.718,)                        # Singleton
>>> t = 'toto',                         # Singleton

>>> v = (,)                            # Tuple vide
>>> w = tuple()                        # Tuple vide
```

Les tuples (2/2)

Comme les strings, les tuples sont immutables !

```
>>> mon_tuple = (1, 2)
>>> mon_tuple[1] = 3          # Attention : on ne peut pas modifier un tuple !
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

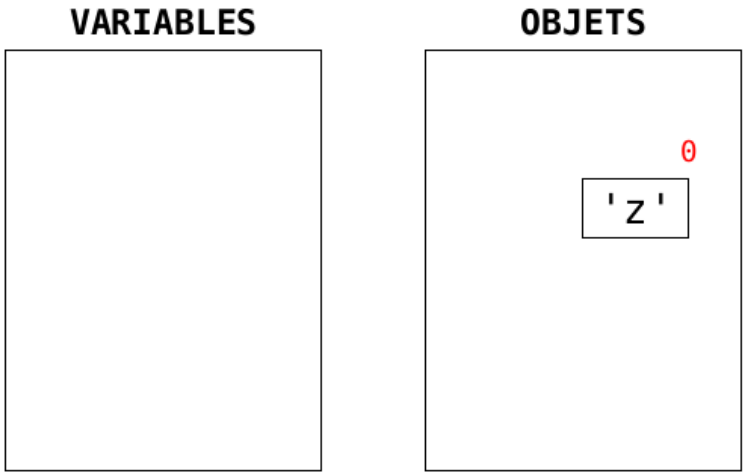
Les listes

Collection hétérogène d'éléments séparés par des virgules, et entourée de crochets

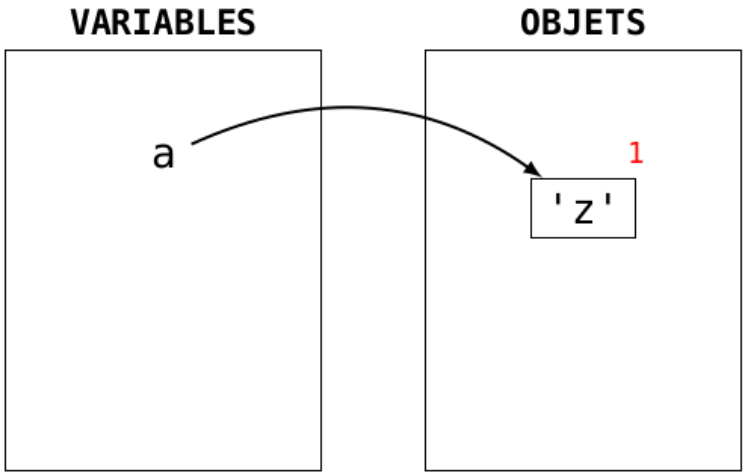
Plan

- Conteneurs standard
  - Les séquences
  - Retour sur les références
  - Les dictionnaires
  - Les ensembles

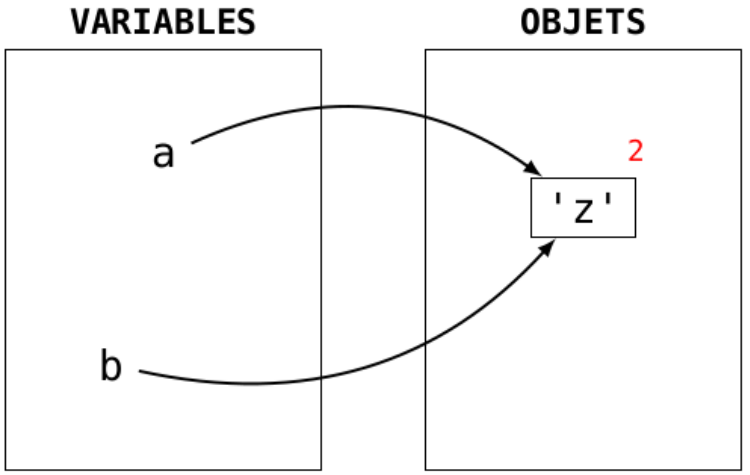
Références partagées des objets immutables



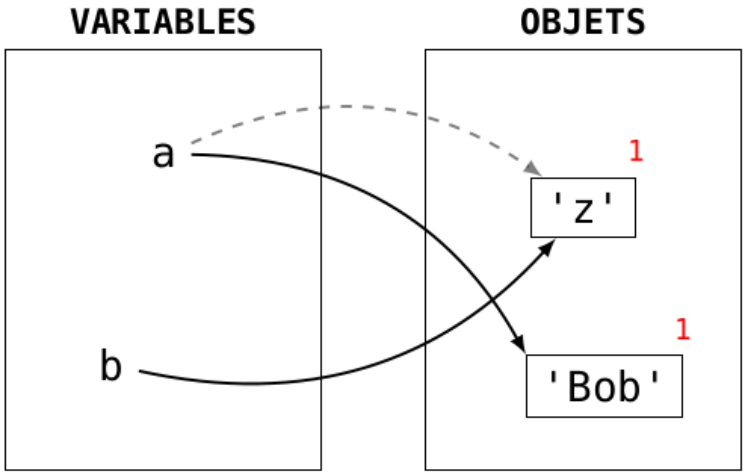
(a) Création d'un objet : `>>> 'z'`



(b) Affectation : `>>> a = 'z'`

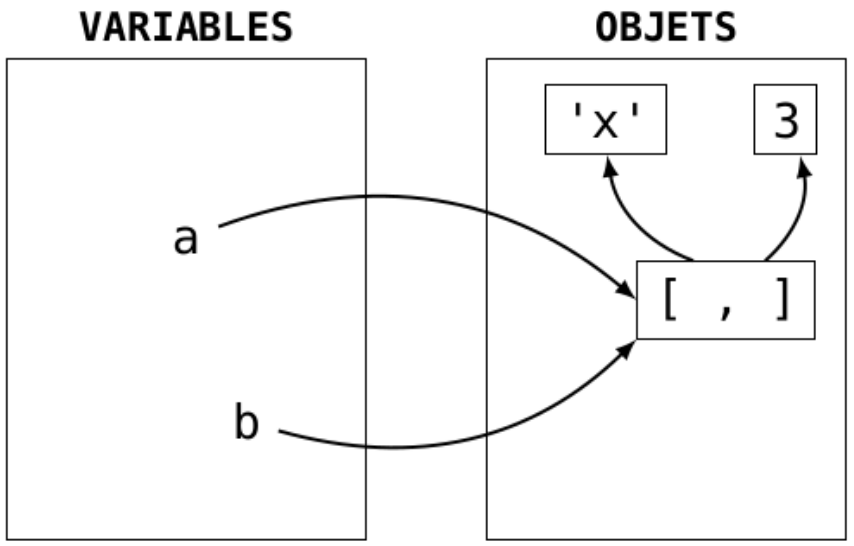


(c) Une référence partagée : `>>> b = a`

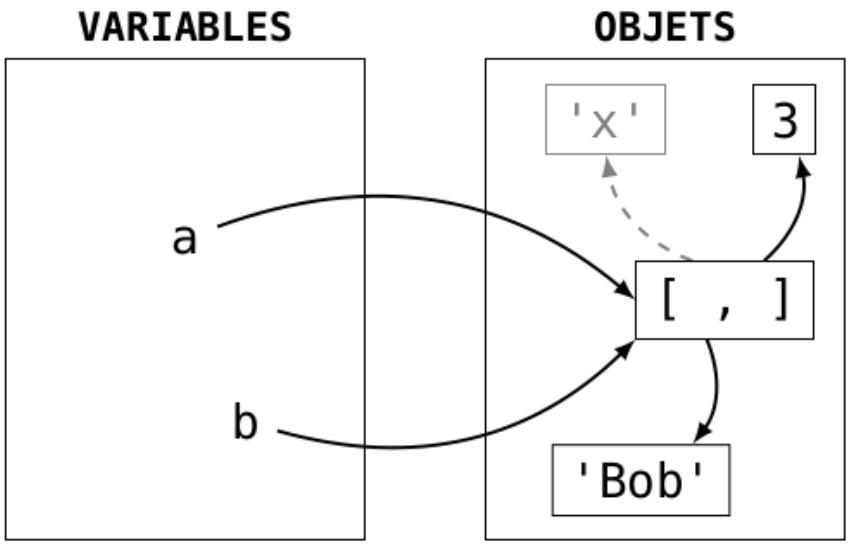


(d) Une référence par objet : `>>> a = 'Bob'`

Références partagées des objets mutables

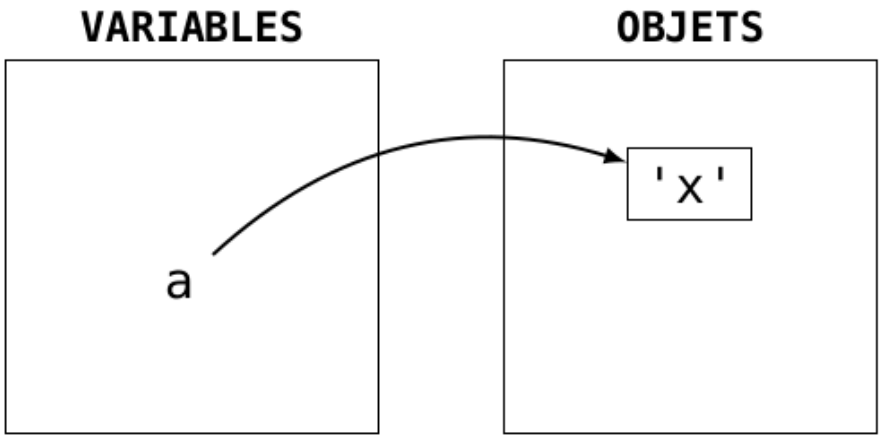


(a) Référence partagée : `>>> b = a`



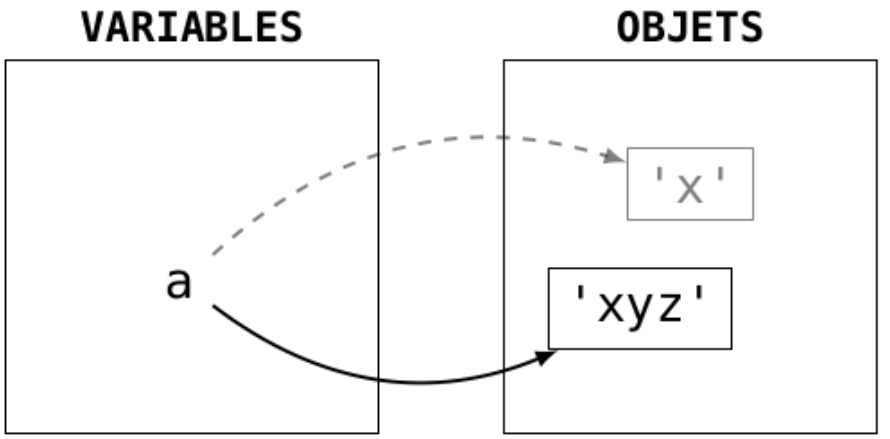
(b) `a` et `b` sont modifiées : `>>> a[0] = 'Bob'`

Affectation augmentée – objet immutable



(a) Affectation d'une chaîne de caractères

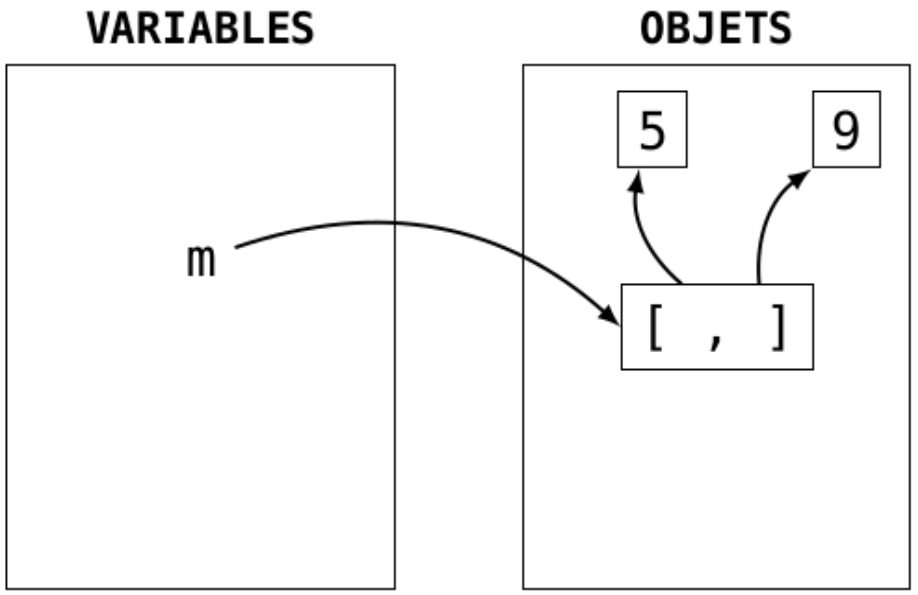
```
>>> a = 'x'
```



(b) Affectation augmentée de la chaîne

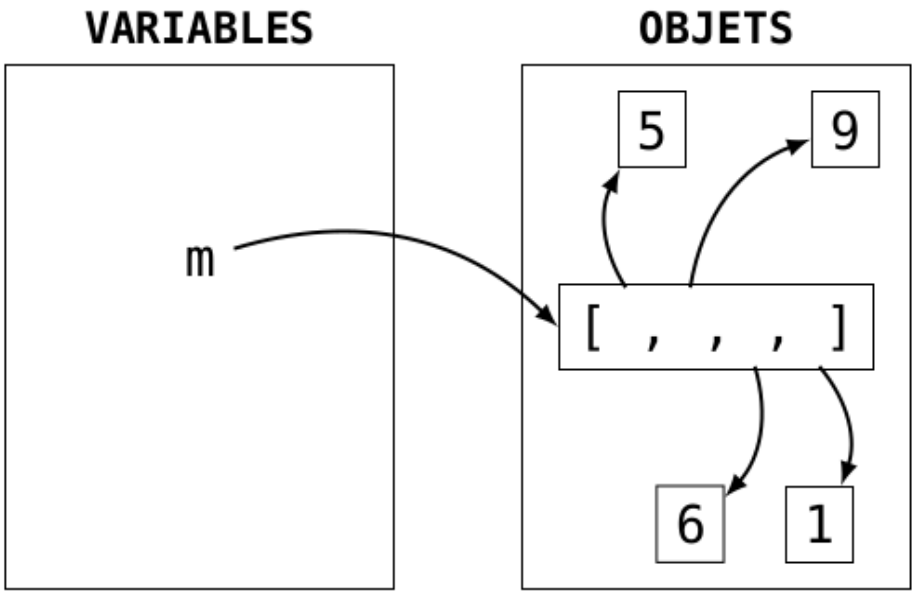
```
>>> a += 'yz'
```

Affectation augmentée – objet mutable



(a) Affectation d'une liste

```
>>> m = [5, 9]
```



(b) Affectation augmentée

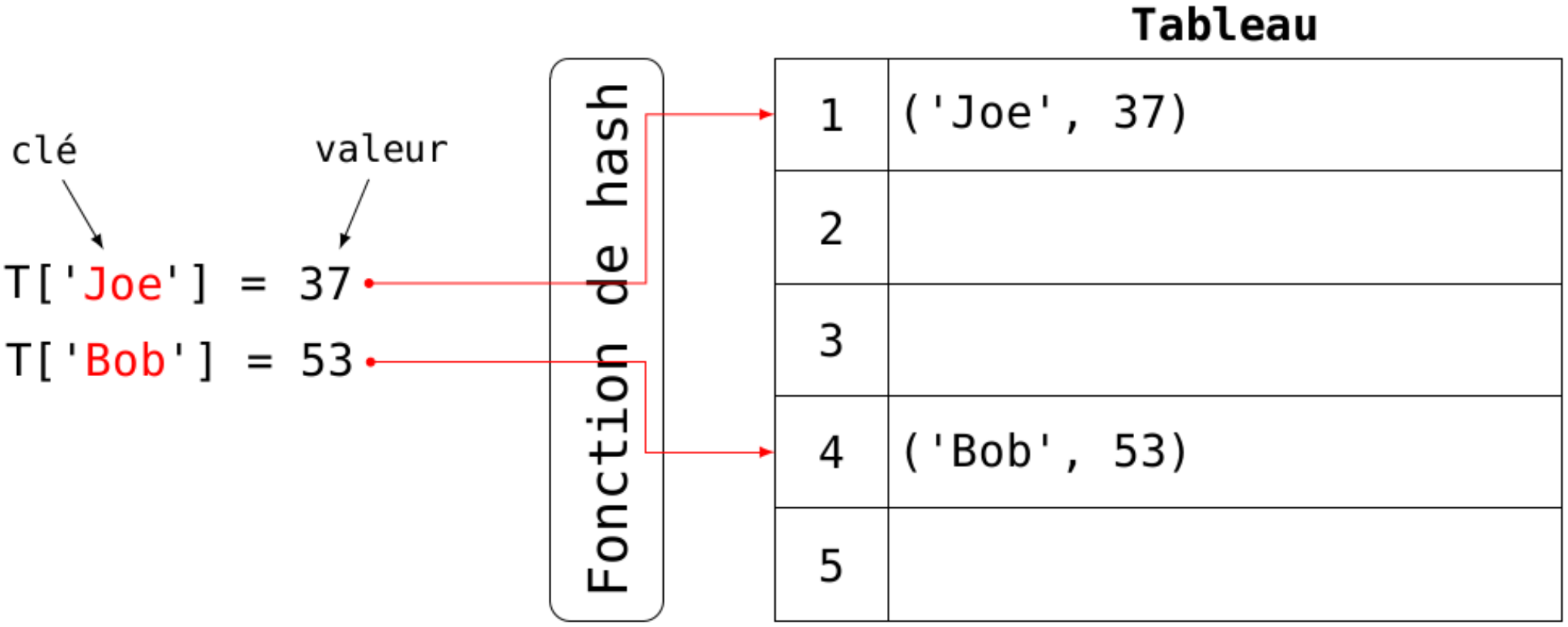
```
>>> m += [6, 1]
```

Plan

- Conteneurs standard
  - Les séquences
  - Retour sur les références
  - Les dictionnaires
  - Les ensembles

Table de hash

C’est un conteneur non ordonné d’éléments indexés par des clés, avec un accès très rapide à un élément à partir de sa clé qui est unique.



Dictionnaires

Techniquement, c'est une table de *hash* mutable dont les clés sont immutables. Plus simplement c'est un **tableau associatif**.

```
>>> dico = {}          # dictionnaire vide. Ou encore : dico =
  dict()

>>> dico['mi'] = 'je'

>>> dico['ŝi'] = 'elle'

>>> dico['vi'] = 'vous'

>>> print(dico)

{'mi':'je', 'ŝi':'elle', 'vi':'vous'}

>>> print(dico['mi'])

'je'

>>> del dico['mi']

>>> print(dico)

{'ŝi':'elle', 'vi':'vous'}
```

Méthodes (1/2)

```
>>> tel = {'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel
{'guido': 4127, 'jack': 4098, 'sape': 4139}  # Un dictionnaire n'est pas ordonné
>>> tel['jack']                              # Valeur de la clé 'jack'
4098
>>> del tel['sape']                          # Suppression d'un coupe (clé : valeur)
>>> tel.keys()                               # Clés de tel
dict_keys(['jack', 'guido'])
>>> tel.values()                             # Valeurs de tel
dict_values([4098, 4127])
>>> 'guido' in tel, 'jack' not in tel        # Teste l'appartenance d'une clé au dictionnaire
(True, False)
```

Méthodes (2/2)

Vue : objet itérable, mis à jour en même temps que le dictionnaire

```
>>> sac = {3:'pommes', 8:'yaourts', 4:'jambon', 1:'pain'}
>>> sac.items()
dict_items([(3, 'pommes'), (8, 'yaourts'), (4, 'jambon'), (1, 'pain')])
>>> sac.values()
dict_values(['pommes', 'yaourts', 'jambon', 'pain'])
>>> k = sac.keys()              # k est une vue
>>> k
dict_keys([3, 8, 4, 1])
>>> del sac[8]                  # Suppression d'un couple
>>> sac[7] = 'beurre salé'      # Ajout d'un couple
>>> k                           # k est automatiquement mis à jour
dict_keys([3, 4, 1, 7])
>>> 8 in k, 7 in k              # Les vues supportent les tests d'appartenance
(False, True)
```

Plan

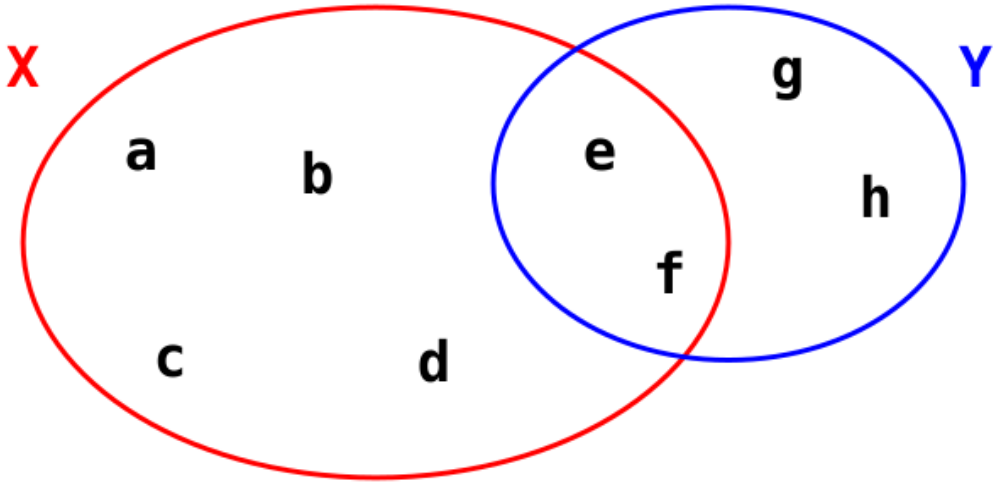
- Conteneurs standard
  - Les séquences
  - Retour sur les références
  - Les dictionnaires
  - Les ensembles

Ensembles (1/2)

Tables de *hash* qui ne stockent que des clés.

```
>>> couleurs = {'trefle', 'carreau', 'coeur', 'pique'}  # Expression littérale
>>> chiffres = set(range(10))        # Construction à partir des éléments d'un itérable
>>> couleurs
{'coeur', 'trefle', 'pique', 'carreau'}
>>> chiffres
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Ensembles (2/2)




```
X = set('abcdef')   # X = {'a', 'b', 'c', 'd', 'e', 'f'}
Y = set('efghf')    # Y = {'e', 'f', 'g', 'h'} pas de duplication : qu'un seul 'f'
'b' in X, 'c' in Y  # (True, False)
X - Y  # {'a', 'b', 'c', 'd'} ensemble des éléments de X qui ne sont pas dans Y
Y - X  # {'g', 'h'} ensemble des éléments de Y qui ne sont pas dans X
X | Y  # {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'} union
X & Y  # {'e', 'f'} intersection
X ^ Y  # {'a', 'b', 'c', 'd', 'g', 'h'} ensemble des éléments qui sont soit dans X, soit dans Y
```

Interlude


SPAM
GOLDEN HONEY GRAIL
Wicked-awesome SPAMALOT™ Collector's edition
NET WT 12 OZ (340g)
Hormel Foods

Le zen de Python : `>>> import this`

Préfère la beauté à la laideur,
l'explicite à l'implicite,
le simple au complexe,
le complexe au compliqué,
le déroulé à l'imbriqué,
l'aéré au compact.
Prends en compte la lisibilité.
Les cas particuliers ne le sont jamais assez pour violer les règles.
Mais, à la pureté, privilégie l'aspect pratique.
Ne passe pas les erreurs sous silence,
ou bâillonne-les explicitement.
Face à l'ambiguïté, à deviner ne te laisse pas aller.
Sache qu'il ne devrait y avoir qu'une et une seule façon de procéder,
même si, de prime abord, elle n'est pas évidente, à moins d'être Néerlandais.
Mieux vaut maintenant que jamais.
Cependant jamais est souvent mieux qu'immédiatement.
Si l'implémentation s'explique difficilement, c'est une mauvaise idée.
Si l'implémentation s'explique aisément, c'est peut-être une bonne idée.
Les espaces de noms, sacrée bonne idée! Faisons plus de trucs comme ça!

Conteneurs standard - exercices

Conteneurs standard

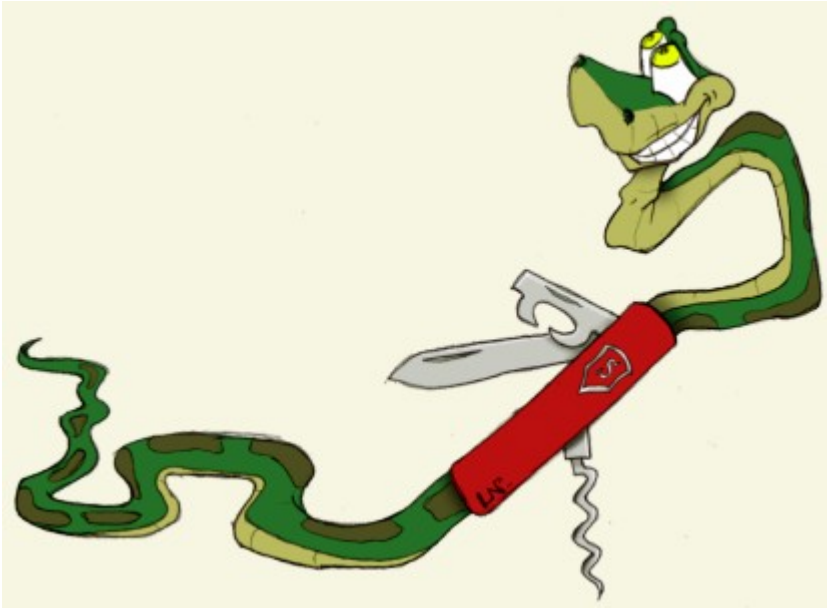
@F_R

- Les séquences ont des propriétés communes.
- Nous avons détaillé les séquences :
  - les listes (mutables),
  - les tuples (immutables).
- Les références partagées des objets mutables peuvent provoquer des effets de bord.
- Les dictionnaires (`dict`) et les ensembles (`set`) permettent des accès en un temps constant.

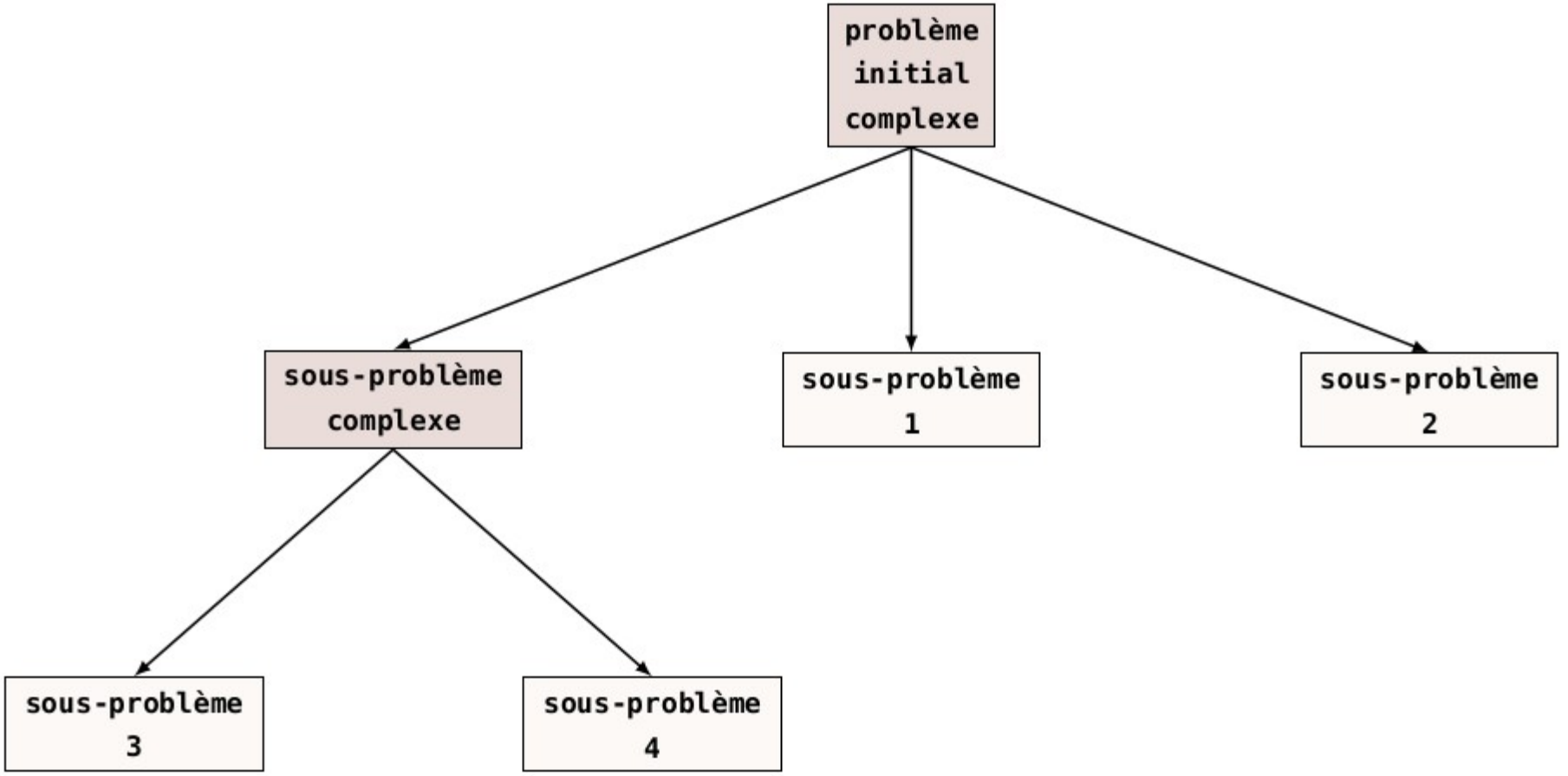
Chapitre 5/11

Fonctions & espaces de nommage

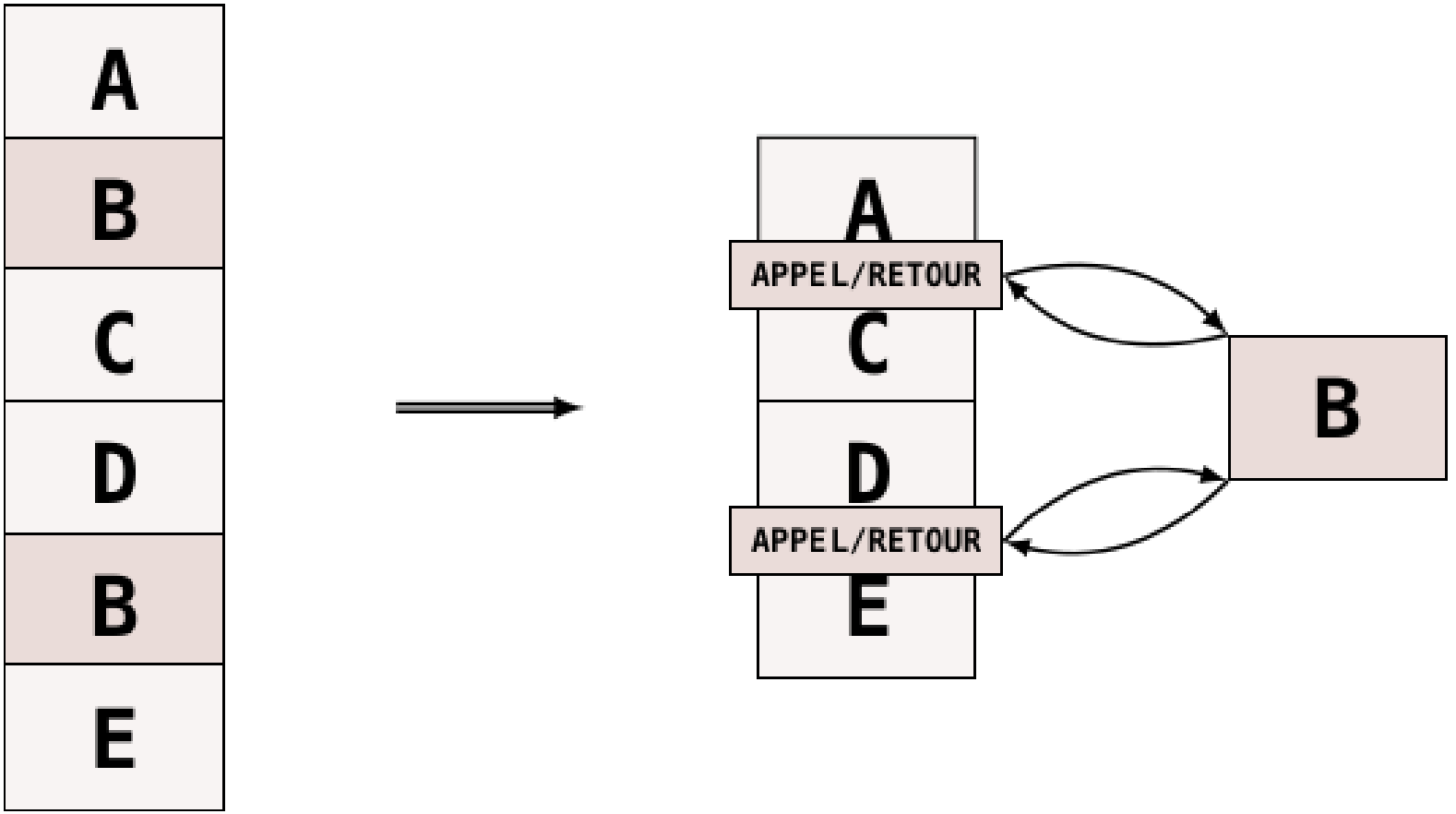
- Définition - exemples de fonctions
- Passage des arguments
- Espaces de nommage



Conception du code



Problème de duplication du code



Fonctions : définition

- Ensemble d'instructions regroupées sous un *nom* et s'exécutant à la demande (*appel*)

Syntaxe :

```
def nom_fonction(paramètres):
    [docstring]
    instructions
```

- Le bloc *instructions* est obligatoire (s'il ne fait rien, on emploie l'instruction `pass`)
- La documentation (*docstring*), bien que facultative, est fortement conseillée

Fonctions : exemples

```
def volume_ellipsoide(a, b, c):  # définition
    """Retourne le volume d'un ellipsoïde de demi-grands axes a, b, c."""
    return 3.14 * a * b * c * 4 / 3

print(volume_ellipsoide(3, 2, 2)) # appel ici ou d'un autre fichier

def divide(a, b):   # définition
    """Fonction protégée par une exception."""
    try:
        return a / b
    except ZeroDivisionError:
        return None

print(divide(19, 7)  # appel
```

« docstring »
Documentation intégrée au code

- On utilise `help(ajouter)` ou `ajouter?` avec ipython :

```
In [4]: def ajouter(a, b):
   ...:     """Retourne la somme des paramètres."""
   ...:     return a + b
   ...:

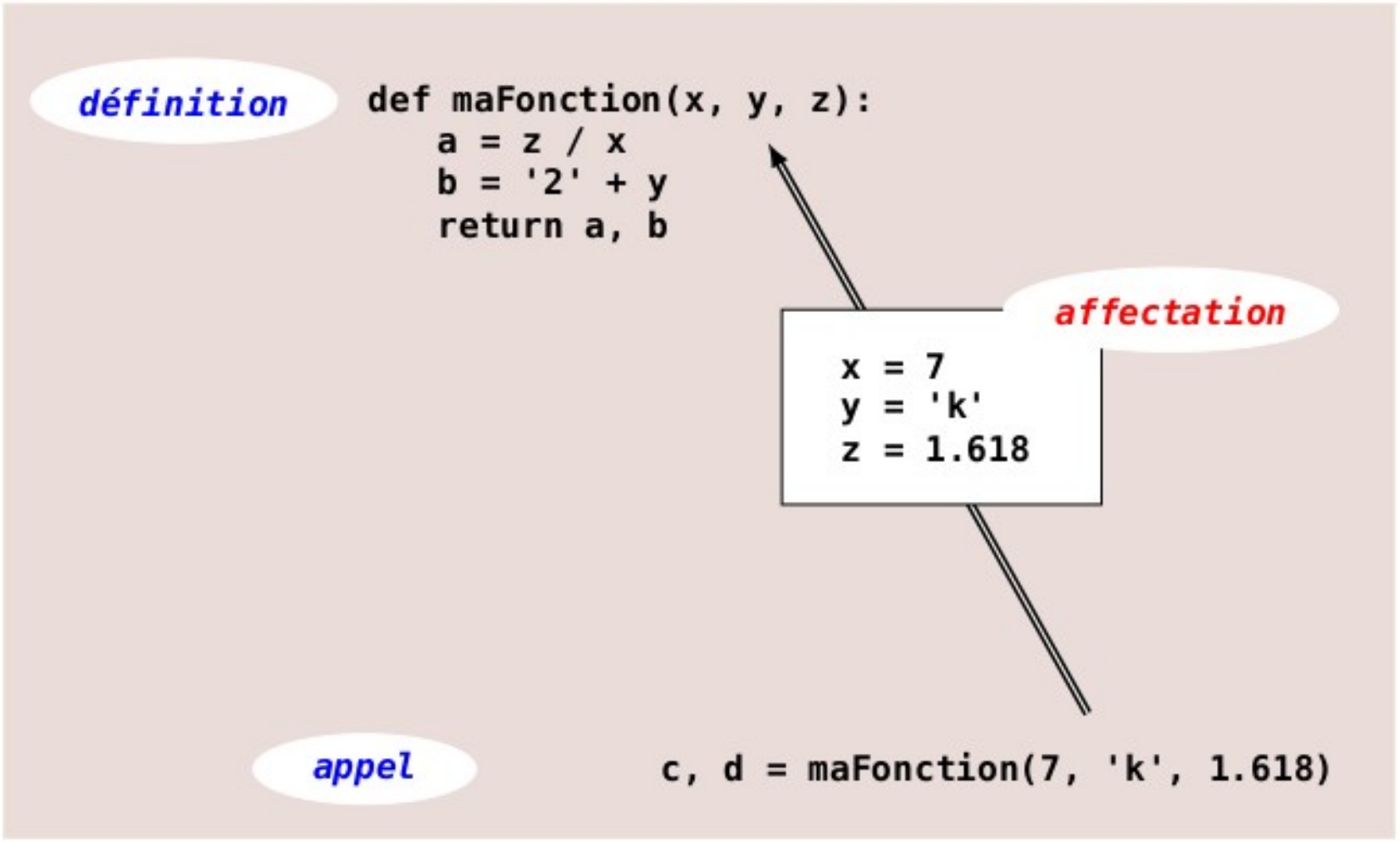
In [5]: ajouter?
Signature: ajouter(a, b)
Docstring: Retourne la somme des paramètres.
File:      /tmp/ipykernel_5035/1599697203.py
Type:      function
```

Plan

Fonctions & espaces de nommage

- Définition - exemples de fonctions
- Passage des arguments
- Espaces de noms

Passage par affectation



Les **arguments d'appel** référencent les **paramètres de définition.**

Un ou plusieurs paramètres positionnels, pas de retour

Sans instruction `return`, on parle parfois de « procédure »

```
def table(base, debut, fin):
    """Affiche la table de multiplication des <base> de <debut> à <fin>."""
    n = debut
    while n <= fin:
        print(n, 'x', base, '=', n * base)
        n += 1

# Exemple d'appel
table(7, 2, 8)
# 2 x 7 = 14 3 x 7 = 21 4 x 7 = 28 5 x 7 = 35 6 x 7 = 42 7 x 7 = 49 8 x 7 = 56
```

Un ou plusieurs paramètres positionnels, un ou plusieurs retours

```
import math

def surfaceVolumeSphere(r):
    surf = 4.0 * math.pi * r**2
    vol = surf * r/3
    return surf, vol

# Programme principal ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
rayon = float(input('Rayon : '))
s, v = surfaceVolumeSphere(rayon)
print("Sphère de surface {:g} et de volume {:g}".format(s, v))
```

Passage d'une fonction en paramètre

Une fonction est un objet de « premier ordre »

```
>>> def double_filtre(lst, fct_filtre):
...     lres = []
...     for v in lst:
...         if fct_filtre(v):
...             lres.append(v * 2)
...     return lres
...
>>> def fgrand(n):
...     return n>10
...
>>> double_filtre(range(1, 20, 3), fgrand)
[26, 32, 38]
>>> def fpair(n):
...     return n%2 == 0
...
>>> double_filtre(range(1, 20, 3), fpair)
[8, 20, 32]
```

Paramètres avec valeur par défaut

On utilise de préférence des valeurs par défaut immutables (types `int`, `float`, `str`, `bool`, `tuple`...) car la modification d'un paramètre par un premier appel est visible les fois suivantes (effet de bord).

```
>>> def accueil(nom, prenom, depart="MP", semestre="S2"):
...     print(prenom, nom, "Département", depart, "semestre", semestre)
...
>>> accueil("Deuf", "John")
John Deuf Département MP semestre S2
>>> accueil("Paradise", "Eve", "Info")
Eve Paradise Département Info semestre S2
>>> accueil("Annie", "Steph", semestre="S3")
Steph Annie Département MP semestre S3
```

Nombre d'arguments arbitraire
(notion avancée)

Utilisation de la notation d'un paramètre final `*<nom_parametre>` appelé par convention `args`.

```
def f(*args):
    print(args)

# Exemples d'appel :
f()             # ()
f(1)            # (1,)
f(1, 2, 3, 4)   # (1, 2, 3, 4)
```

« Décapsulation »
(notion avancée)

Réciproquement, il est aussi possible de passer une séquence à l'appel, qui sera **décapsulée** en une liste de paramètres ordonnés.

```
def somme(a, b, c):
    return a+b+c

# Exemple d'appel :
elements = (2, 4, 6)
print(somme(*elements))     # 12
```

Nombre d'arguments arbitraire : passage d'un dictionnaire
(notion avancée)

```
def unDict(**kwargs):
    return kwargs

# Exemples d'appels
## par des paramètres nommés :
print(unDict(a=23, b=42))      # {'a': 23, 'b': 42}

## en fournissant un dictionnaire :
mots = {'d': 85, 'e': 14, 'f':9}
print(unDict(**mots))          # {'e': 14, 'd': 85, 'f': 9}
```

Fonctions *variadiques* : `*t, **d`

La définition de fonction :

```
def f(*t, **d) :
      pass
```

est une définition **générique** qui supporte tout type d'appel (paramètres positionnels, paramètres nommés…)

Passage d'un argument mutable
Danger !

```
# Opération avec paramètre mutable
def ajoute_1(x):
    x.append(1)
    return x


lst = [1, 4, 5]

print(ajoute_1(lst))      # [1, 4, 5, 1]
print(lst)                # [1, 4, 5, 1]    (lst est modifié à chaque appel)
print(ajoute_1(lst))      # [1, 4, 5, 1, 1]
print(lst)                # [1, 4, 5, 1, 1]
print(ajoute_1(lst))      # [1, 4, 5, 1, 1, 1]
print(lst)                # [1, 4, 5, 1, 1, 1]
```

Plan

Fonctions & espaces de nommage

- Définition - exemples de fonctions
- Passage des arguments
- Espaces de nommage

Espaces de nommage
(chacun chez soi)

Il permet de lever une ambiguïté sur des objets qui pourraient être homonymes. Il est matérialisé par un préfixe identifiant de manière unique leur origine. Dans l'exemple suivant, les trois méthodes open( ) appartiennent à des packages différents :

```
>>> import webbrowser, os, PIL.Image
>>> webbrowser.open("http://www.dunod.fr")
True
>>> os.open("/etc/hosts", os.O_RDONLY)
4
>>> PIL.Image.open("../figs/chap5_r.png")
<PIL.PngImagePlugin.PngImageFile image mode=RGBA size=366x519 at 0x7F9D65060E10>
```

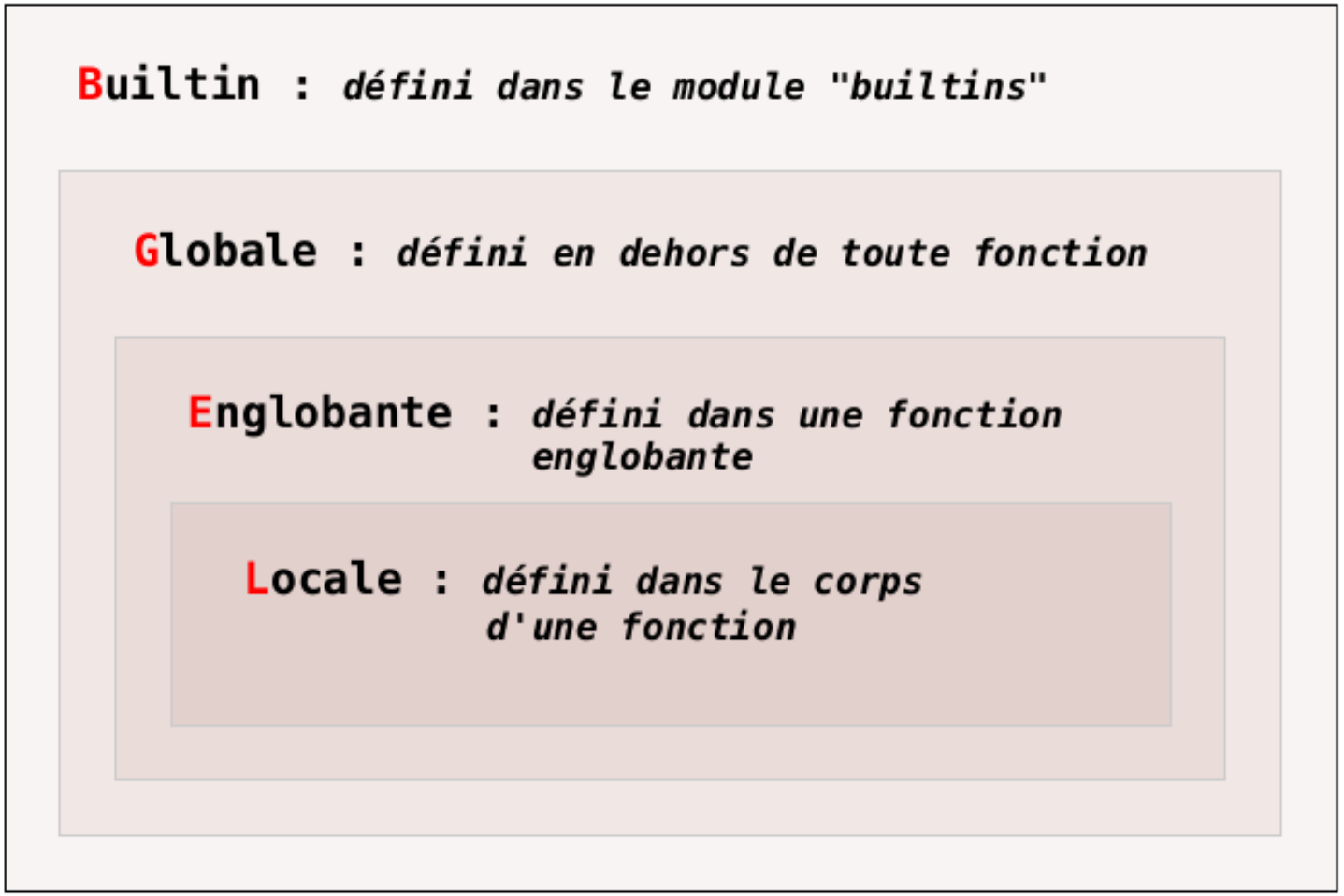
Portée des objets

Les noms des objets sont créés lors de leur **première affectation**. Ils ne sont visibles que dans certaines régions de la mémoire :

- Portée globale : celle du module **`__main__`**. Un dictionnaire gère les objets globaux : l'instruction **`globals()`** fournit les couples **`variable:valeur`**
- Portée locale : les objets internes aux fonctions (et aux classes) sont locaux. Les objets globaux ne sont pas modifiables dans les portées locales. L'instruction **`locals()`** fournit les couples **`variable:valeur`**

Résolution des noms : règle *LEGB*

La recherche des noms est d'abord locale L, puis englobante E, puis globale G, enfin builtin B



Les portées (*scope*)

Fonctions & espaces de nommage exercices

Fonctions et espaces de nommage

@F$_R$

Usuellement on distingue :

- deux manières de définir les paramètres d'une fonction :
  - ordonnée,
  - valeur par défaut ;
- deux méthodes de passage des arguments :
  - ordonnée,
  - nommée.

On dispose en outre des formes « étoilées », qui permettent :

- de définir des paramètres recevant un tuple ou un dictionnaire ;
- le passage par « décapsulation » d'un tuple ou d'un dictionnaire.

Les espaces de nommage régissent la visibilité des objets. Les noms ayant comme portée la fonction n'existent que le temps de l'appel à celle-ci.

Rappels sur le passage des arguments

- ```
  def f(x, *t) :  # param. pos. + tuple
      …
  f(1, 2, 3, 4)  # x → 1, t → (2, 3, 4)
  ```

- ```
  def g(x, y, **d) :  # param. pos. + dictionnaire
      …
  g(2, 3, flag=True, header='debug')
  # x → 2, y → 3, d → {'flag':True, 'header':'debug'}
  ```

- ```
  def h(*t, **d) :  # définition générique
      …
  ```
  ```

Thonny & Jupyterlab

Chapitre 6/11

Modules & packages

- Module
- Packages

Modules

Fichiers contenant une collection d'outils apparentés réutilisables (fonctions, classes, données…).

Un module (ou bibliothèque) est un espace de noms **mutable**.

Un module permet :

- la réutilisation du code
- la documentation et les tests unitaires peuvent être intégrés au module

La commande `import`

- `import nom_module` : donne accès à l'ensemble des définitions du module importé en utilisant le nom du module comme espace de noms :

```
>>> import tkinter
>>> print("Version de tkinter :", tkinter.TkVersion)
Version de tkinter : 8.6
```

- `from nom_module import nom1, nom2...`

  donne accès directement à une sélection choisie de noms définis dans le module :

```
>>> from math import pi, sin
>>> print("Valeur de Pi :", pi, "sinus(pi/4) :", sin(pi/4))
Valeur de Pi : 3.14159265359 sinus(pi/4) : 0.707106781187
```

- `import numpy as np` (alias adopté par la communauté)

Syntaxe

- On omet l'extension `.py` : `import math`
- Les librairies partagées (`.so`, `.dll`, `.dylib...`) construites à partir de sources en C, en C++ (Cython), en FORTRAN... sont directement utilisables
- La PEP 8 conseille d'importer dans l'ordre :
  - les modules de la bibliothèque standard, puis leur contenu
  - les modules tierce partie, puis leur contenu
  - les modules du projet, puis leur contenu

Auto-test

- La valeur de la variable globale réservée `__name__` spécifique à chaque module nous permet d'identifier le module principal. Dans le module en cours d'exécution, sa valeur sera égale à `"__main__"`. Dans un autre, il vaudra le nom du fichier
- Il est alors possible de placer du code conditionnel à l'initialisation d'un module qui ne sera exécuté que si celui-ci est le module principal (non importé).

- Le module a donc la structure suivante :
  - en-tête
  - définition des globales / constantes
  - définition des fonctions et / ou classes
  - code conditionnel d'auto-test

Auto-test (avec spyder)

Modules
(exercices avec spyder)

Plan

- Modules & packages
  - Module
  - Packages

Packages

- C’est un module contenant d’autres modules
- Les modules d’un package peuvent être des sous-packages, ce qui donne une structure arborescente

Par exemple :

```
malib/
     __init__.py
     calculs.py
     affich.py
     stockage/
          __init__.py
          disque.py
          reseau.py
          commun.py
```

Avec les fichiers :

```
# Fichier : malib/__init__.py
VERSION=1

# Fichier : malib/calculs.py
def som(a, b):
    return a + b

# Fichier : malib/stockage/__init__.py
DEFAUT="disque"
```

Utilisation

```
from malib import VERSION
import malib.stockage
print(f"Version: {VERSION}, stockage dans: {malib.stockage.DEFAUT}")
```

Modules et packages

@F_R

- La programmation multi-fichiers permet de structurer son code.
- Comprendre les mécanismes de l'importation des modules.
- Utiliser les « auto-tests » pour valider ses modules.
- Savoir organiser et utiliser un package.

Chapitre 7/11

Accès aux données

- Fichiers
- Gérer les fichiers et les répertoires
- Sérialisation
- Bases de données relationnelles

Fichiers

- "file" est un **type prédéfini** de Python
- Les données utilisées dans la mémoire d'un ordinateur sont temporaires. Pour les stocker de façon permanente on doit les enregistrer dans un fichier sur un disque ou sur un autre périphérique de stockage permanent (disque dur, clé USB, DVD...)
- La persistance consiste à sauvegarder des données afin qu'elles survivent à l'arrêt de l'application. On doit assurer le stockage et le rapatriement des données
- Noms des fichiers et des répertoires : on évitera en général les caractères `\ / * ? < > " | :`

Ouverture et fermeture des fichiers en mode texte

- Python ouvre les fichiers en mode texte par défaut (mode `"t"`). Pour les fichiers binaires, il faut préciser explicitement le mode `"b"` (par exemple : `"wb"` pour une écriture en mode binaire).

- Principaux modes d'ouverture :

```
f1 = open("monFichier_1", "r", encoding='utf8') # en lecture
f2 = open("monFichier_2", "w", encoding='utf8') # en écriture
f3 = open("monFichier_3", "a", encoding='utf8') # en ajout
```

- Une seule méthode de fermeture :

```
f1.close()
```

Écriture séquentielle

- Méthodes d'écriture :

```
f = open("truc.txt", "w", encoding='utf8')
s = 'toto\n'
f.write(s)                  # Écrit la chaîne s dans f
l = ['a', 'b', 'c']
f.writelines(l)             # Écrit les chaînes de la liste l dans f
f.close()
```

- Ce qui produit l'enregistrement :

```
Fichier truc.txt :
toto
abc
```

Lecture séquentielle

Méthodes de lecture :

```
f = open("truc.txt", "r", encoding='utf8')
s1 = f.read()          # lit tout le fichier -> string
s2 = f.read(n)         # lit au plus n octets -> string
s3 = f.readline()      # lit la ligne suivante -> string
lst = f.readlines()    # lit tout le fichier -> liste de strings
```

Syntaxe fréquente :

```
for ligne in f:
    print(ligne)       # bon procédé de parcours d'un fichier
f.close()
```

Gestionnaire de contexte `with`

Utiliser une ressource dans un bloc de code puis terminer en fermant proprement l'accès (que l'on sorte de ce bloc normalement ou suite à une exception) est un motif **récurrent**. L'instruction `with` gère élégamment ce cas de figure.

```
with open(filename, encoding='utf8') as fh:
    for line in fh:
        process(line)
```

Fichiers binaires

- Dans certains domaines (par exemple acquisitions de données de manips) il est fréquent de gérer des fichiers binaires, plus compacts que les fichiers textuels.
- Dans l'exemple suivant, on ouvre le fichier binaire `'data.bin'` en mode écriture `'bw'`. Dans lequel on écrit 500 fois le caractère `'é'` (codé `b'\xe9'` en hexadécimal) en mode byte (préfixe `b, type bytes`) :

```
with open('data.bin', 'bw') as f:
    f.write(b'\xe9' * 500)
```

Les fichiers

Plan

Accès aux données

- Fichiers
- Gérer les fichiers et les répertoires
- Sérialisation
- Bases de données relationnelles

Gérer les fichiers et les répertoires

Plan

Accès aux données

- Fichiers
- Gérer les fichiers et les répertoires
- Sérialisation
- Bases de données relationnelles

Sérialisation `pickle`

Le module `pickle` est simple mais confiné à Python.

```python
import pickle

favorite_color = {"lion": "jaune", "fourmi": "noire", "caméléon": "variable"}
# Stocke ses données dans un fichier
with open("pickle_tst", "bw") as f:
    pickle.dump(favorite_color, f)

# Retrouver ses données : pickle recrée un dictionnaire
with open("pickle_tst", "br") as f:
    dico = pickle.load(f)
print(dico)
```

L'affichage des données relues produit :

```
{'lion': 'jaune', 'fourmi': 'noire', 'caméléon': 'variable'}
```

Sérialisation `json`

`json` est un format d'échange standardisé implémenté dans un grand nombre de langages.

```
import json

# Encodage dans un fichier
with open("json_tst", "w") as f:
    json.dump(['foo', {'bar':('baz', None, 1.0, 2)}], f)

# Décodage
with open("json_tst") as f:
    print(json.load(f))
```

Le fichier `json_tst` contient :

```
['foo', {'bar': ['baz', None, 1.0, 2]}]
```

Plan

Accès aux données

- Fichiers
- Gérer les fichiers et les répertoires
- Sérialisation
- Bases de données relationnelles

Bases de données relationnelles

Pour mémoire :

- ✔ Un SGBDR (Systèmes de Gestion de Bases de Données Relationnelles) est un logiciel de stockage et de manipulation de données
- ✔ Il est programmable *via* le langage SQL
- ✔ La bibliothèque standard Python fournit le SGBDR `SQLite3`

Accès aux données

@F$_R$

- Gestion des fichiers en mode texte.
- Les facilités d'emploi du gestionnaire de contexte `with`.
- Gestion des fichiers binaires.
- Gestion des fichiers et des répertoires.
- La sérialisation.
- Notions de langage SQL.
- Compréhension de ce que recouvrent Internet et ses protocoles.

Chapitre 8/11

La POO

- Classes et instanciation d'objets
- Méthodes
- Méthodes spéciales
- Héritage et polymorphisme
- Espaces de nommage

La POO : motivations

- La POO permet de mieux **modéliser** la réalité en concevant des modèles d'objets, les classes
- Ces classes permettent de construire des **objets interactifs** entre eux et avec le monde extérieur
- Les objets sont créés indépendamment les uns des autres, grâce à **l'encapsulation**, mécanisme qui permet d'embarquer leurs propriétés
- Les classes permettent d'**éviter** au maximum l'emploi des **variables globales**
- Enfin les classes offrent un moyen économique et puissant de construire de **nouveaux objets** à partir d'objets **préexistants**.

Terminologie

- Une **classe** est une famille d'objets équivalente à un nouveau type de données. Par exemple les classes `list` ou `str` et leurs nombreuses méthodes permettant de les manipuler en utilisant la notation pointée :

```
In [1]: lst = [3, 5, 1, 9]

In [2]: lst.remove(5)

In [3]: lst
Out[3]: [3, 1, 9]

In [4]: "casse".upper()
Out[4]: 'CASSE'
```

- Un **objet** ou une **instance** est un exemplaire particulier d'une classe

Terminologie (suite)

- Les objets ont donc généralement deux sortes d'attributs : les données, nommées simplement **attributs**, et les fonctions applicables, appelées **méthodes**
- Par exemple un objet de la classe `complex` possède :
  - deux attributs : `imag` et `real` ;
  - plusieurs méthodes, comme `conjugate()`, `abs()`...
- La plupart des classes **encapsulent** à la fois les données et les méthodes applicables à leurs objets
  - Par exemple un objet `str` contient une chaîne de caractères Unicode (les données) et de nombreuses méthodes
- On peut définir un objet comme une **capsule** contenant des attributs et des méthodes :

objet = attributs + méthodes

Syntaxe : instruction `class`

- Les classes sont des **fabriques d'objets**. En utilisant cette métaphore, on construit d'abord l'usine *avant* de produire des objets !
- Définir une nouvelle classe revient exactement à définir un nouveau type de données
- **Syntaxe** : `class` est une instruction composée. Elle comprend un en-tête (avec *docstring*) et un corps indenté :

```
class MaClasse:
    """Documentation de la classe MaClasse."""
    # Définition de la classe : attributs de classe, méthodes
    attcl = 5
    def meth(self):
        pass
```

L’instanciation et ses attributs

On instancie un objet (c’est-à-dire qu’on le produit à partir de l’usine) **en appelant sa classe** comme s’il s’agissait d’une fonction :

```
class MaClasse()
    """Docstring."""
    x = 23              # 3 attributs de classe
    y = "un string"
    z = [1, x, 2, y]


a = MaClasse()          # instanciation
print(a.x, a.y, a.z)  # 23, 'un string', [1, 23, 2, 'un string']
```

POO – exemples de base

Plan

La POO

- Classes et instanciation d'objets
- Méthodes
- Méthodes spéciales
- Héritage et polymorphisme
- Espaces de nommage

Les méthodes

Une méthode s'écrit comme une fonction mais avec un premier paramètre *self* **obligatoire** :

```
class C1:

    x = 23

    y = x + 5

    def affiche(self):

        print(C1.y)     # on qualifie les attributs de
          classe


obj = C1()        # instanciation de l'objet "obj"

obj.affiche()     # 28

# à l'appel de la méthode, "self" est remplacé par "obj"
```

Tout ce qui a été dit pour les fonctions est applicable aux méthodes

Plan

La POO

- Classes et instanciation d'objets
- Méthodes
- Méthodes spéciales
- Héritage et polymorphisme
- Espaces de nommage

Les méthodes spéciales

Ces méthodes sont notées d'un nom prédéfini, précédé et suivi de deux caractères de soulignement.

Par exemple :

```
__init__()
__str__()
```

Elles servent :

- à initialiser l'objet instancié
- à modifier son affichage
- à surcharger ses opérateurs
- ...

La méthode `__init__()`

Lors de l'instanciation d'un objet, ce *constructeur* est automatiquement invoqué. Il permet d'effectuer toutes les initialisations nécessaires :

```
class C2:
    def __init__(self, n):
        self.x = n  # on initialise l'attribut d'instance x
        self.y = 3 * n - 2 # autre attribut d'instance

uneInstance = C2(42)   # 1 argument obligatoire
print(uneInstance.x)   # 42
print(uneInstance.y)   # 124
```

```
class Time :
    def __init__(self, hh=12, mm=0, ss=0) :
        self.heure = hh
        self.minute = mm
        self.seconde = ss
    def affiche_heure(self) :
        print(f"{self.heure}:{self.minute}:{self.seconde}")
```

Utilisation :

```
>>> rendezVous = Time(10, 15)
>>> rendezVous.affiche_heure()
10:15:00
```

On **instancie** un objet (c'est-à-dire qu'on le produit à partir de l'usine) en appelant sa classe.

L'objet est créé en mémoire puis le **constructeur** de la classe, `__init__()`, est automatiquement appelé :

(Cf. listing `Seb/src/Exemple_POO/cotisation.py`)

- L'instanciation d'un objet de la classe `F6KRK` nécessite au moins un argument (son nom) car le second est nommé par défaut
- L'appel à l'attribut d'instance (qui n'est pas lié à un objet particulier), nécessite qu'il soit préfixé par le nom de sa classe. Par exemple :

```
F6KRK.nb_membres
```

Surcharge des opérateurs

Permet à un opérateur de posséder un sens pour plusieurs type :

```
x = 7 + 9          # addition entière
c = 'ab' + 'cd'    # concaténation
```

Python possède des méthodes de surcharge pour :

- tout type (**`__call__`**, **`__str__`**, ...)
- nombres (**`__add__`**, **`__div__`**, ...)
- séquences (**`__len__`**, **`__iter__`**, ...)
- ...

Surcharge arithmétiques courantes

Nom	Méthode spéciale	Utilisation
opposé	`__neg__`	`-obj1`
addition	`__add__`	`obj1 + obj2`
soustraction	`__sub__`	`obj1 - obj2`
multiplication	`__mul__`	`obj1 * obj2`
division	`__div__`	`obj1 / obj2`
division entière	`__floordiv__`	`obj1 // obj2`

Exemple de surcharge

```
class Vecteur2D:
    def __init__(self, x, y):
        self.x, self.y = x, y
    def __add__(self, other):      # addition vectorielle
        return Vecteur2D(self.x + other.x, self.y + other.y)
    def __str__(self):             # pour l'affichage
        return f"Vecteur{self.x, self.y}"

v1 = Vecteur2D(1.2, 2.3)
v2 = Vecteur2D(3.4, 4.5)
print(v1 + v2)  # Vecteur(4.6, 6.8)
```

POO - exercices

Plan

La POO

- Classes et instanciation d'objets
- Méthodes
- Méthodes spéciales
- Héritage et polymorphisme
- Espaces de nommage

Définitions

- L'**héritage** est le mécanisme qui permet de se servir d'une classe préexistante pour en créer une nouvelle qui possédera des fonctionnalités supplémentaires ou différentes.

- Le **polymorphisme** par dérivation est la faculté pour deux méthodes (ou plus) portant le même nom, mais appartenant à des classes héritées distinctes, d'effectuer un travail différent. Cette propriété est acquise par la technique de la surcharge.

- (Cf. listing `Seb/src/Exemple_POO/rectangle.py`)

Plan

La POO

- Classes et instanciation d'objets
- Méthodes
- Méthodes spéciales
- Héritage et polymorphisme
- Espaces de nommage

Retour sur les espaces de noms

- On a déjà vu que les espaces obéissaient à la règle LEGB (« Locale Englobante Globale Builtin » qui définit dans quel ordre ces espaces sont parcourus pour résoudre un nom.
- Les classes ainsi que les objets instances définissant de nouveaux espaces de noms, il y a là aussi des règles pour résoudre les noms entre ces espaces :
  - Un nom non qualifié est accessible directement sans notation pointée devant lui
  - Un nom qualifié par une classe est précédé par une notation pointée spécifiant une classe.
  - Un nom qualifié par une instance est précédé par une notation pointée spécifiant une variable d'instance (le paramètre `self` passé aux méthodes est une variable référençant l'instance manipulée dans la méthode).

POO – héritage & polymorphisme

Notion de conception objet

Suivant les relations que l'on va établir entre les objets de notre application, on peut concevoir nos classes en utilisant *l'association* ou la *dérivation*.

L'**association** repose sur la relation *« a-un »* ou *« utilise-un »* et se traduit par une **composition** :

```
class A:
    pass

class B:
    def __ini__(self) :
        self.attr_inst = A()
```

Les objets des classes `A` et `B` peuvent exister **séparément**.

Notion de conception objet (suite)

La **dérivation** repose sur la relation *« est-un »* et se traduit par un **héritage** :

```
class A:
    def __init__(self)
        pass

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

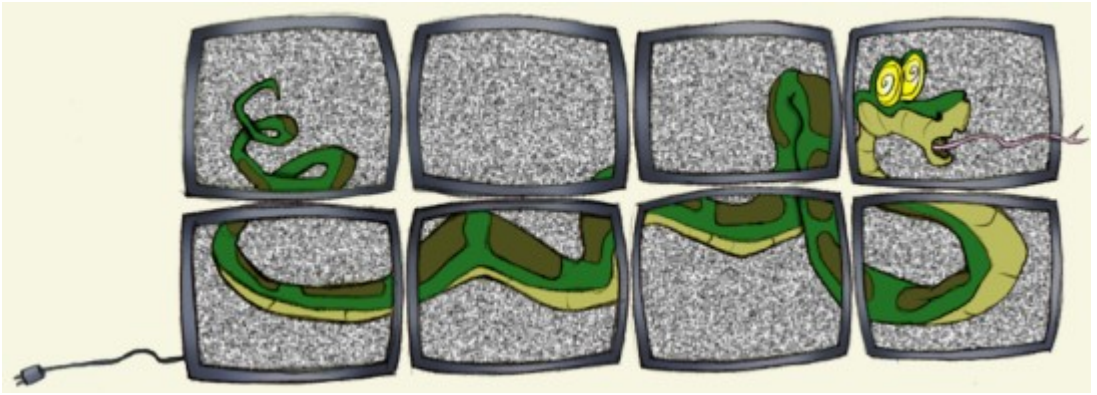
Bien sûr, ces deux conceptions cohabitent souvent !

Programmation orientée objet

- La notion de classe : la fabrique.
- L'*instanciation* : l'objet.
- Attributs et méthodes.
- Le constructeur et les méthodes spéciales.
- Puissance de l'héritage et des surcharges.
- Quelques notions de conception objet.

Chapitre 9/11

La POO graphique



Programmes pilotés par des événements

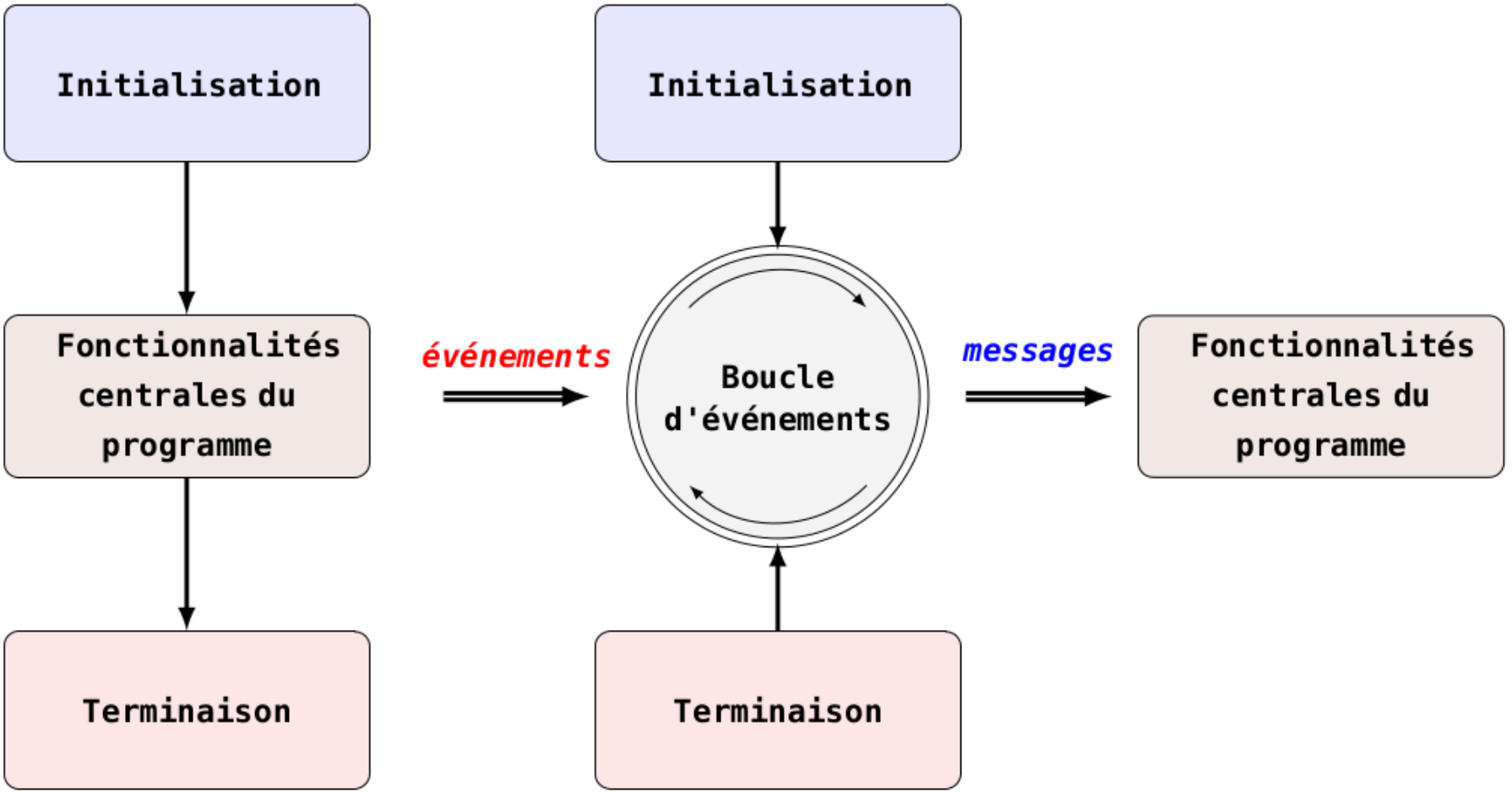
Les bibliothèques graphiques

Les classes de *widgets* de Tkinter

Le positionnement des *widgets*

Exemples de POO graphique

Programmes pilotés par événements



(a) Séquentielle

(b) Pilotée par une boucle d'événements

Plan

La POO graphique

- Programmes pilotés par des événements
- Les bibliothèques graphiques
- Les classes de *widgets* de Tkinter
- Le positionnement des *widgets*
- Exemples de POO graphique

Les bibliothèques graphiques

- Parmi les différentes bibliothèques graphiques utilisables dans Python (**`wxPython`**, **`Qt`**...), la bibliothèque **`Tkinter`**, issue du langage `tcl/Tk` est très employée, car elle est installée de base dans toutes les distributions Python (*batteries included*).

- **`Tkinter`** facilite la construction de GUI simples. Après avoir importé le module, on crée, configure et positionne les *widgets* souhaités, puis on entre dans la boucle principale de gestion des événements.

Plan

La POO graphique

- Programmes pilotés par des événements
- Les bibliothèques graphiques
- Les classes de *widgets* de Tkinter
- Le positionnement des *widgets*
- Exemples de POO graphique

Exemples de base
(Seb/src/POO_graph)

Les widgets de Tkinter (1/3)

`Tk`	fenêtre de plus haut niveau
`Frame`	contenant pour organiser d'autres widgets
`Label`	zone de message
`Button`	bouton avec texte ou image
`Message`	zone d'affichage multi-lignes
`Entry`	zone de saisie
`Checkbutton`	bouton à deux états
`Radiobutton`	bouton à deux états exclusifs

Les widgets de Tkinter (2/3)

`Scale`	glissière à plusieurs positions
`PhotoImage`	pour placer des images sur des widgets
`BitmapImage`	pour placer des bitmaps sur des widgets
`Menu`	associé à un **`Menubutton`**
`Menubutton`	bouton ouvrant un **`Menu`** d'options
`Scrollbar`	ascenseur
`Listbox`	liste de noms à sélectionner
`Text`	édition de texte multi-lignes

Les widgets de Tkinter (3/3)

`Canvas`	zone de dessin graphique ou de photos
`OptionMenu`	composé : liste déroulante
`ScrolledText`	composé : **`Text`** avec un ascenseur
`PanedWindow`	interface à onglets
`LabelFrame`	contenant avec un cadre et un titre
`Spinbox`	un widget de sélection multiple

Documentation officielle :

`python.doctor/page-tkinter-interface-graphique-python-tutoriel`

Plan

La POO graphique

- Programmes pilotés par des événements
- Les bibliothèques graphiques
- Les classes de *widgets* de Tkinter
- Le positionnement des *widgets*
- Exemples de POO graphique

Gestion de la géométrie

Tkinter possède trois gestionnaires :

- Le **`packer`** : dimensionne et place chaque widget dans un widget conteneur selon l'espace requis par chacun d'eux
- Le **`gridder`** : dimensionne et positionne chaque widget dans les cellules d'un tableau dans un widget conteneur
- Le **`placer`** : dimensionne et place chaque widget `w` dans un widget conteneur exactement selon ce que demande `w`. C'est un placement absolu (usage peu fréquent)

Gridder
(Seb/src/POO_graph/verif_tkpasswd.py)

Plan

La POO graphique

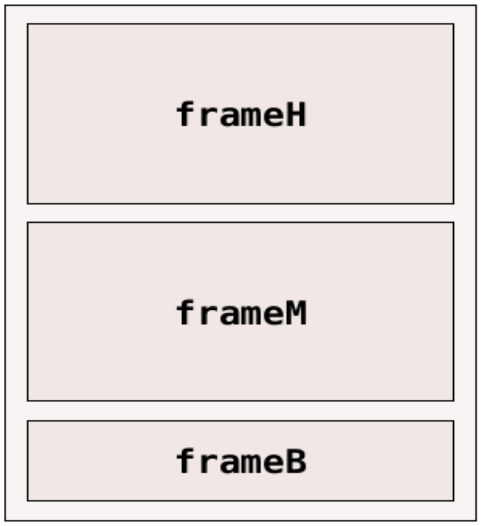
- Programmes pilotés par des événements
- Les bibliothèques graphiques
- Les classes de *widgets* de Tkinter
- Le positionnement des *widgets*
- Exemples de POO graphique

Style POO
(Seb/src/POO_graph/hi.py)

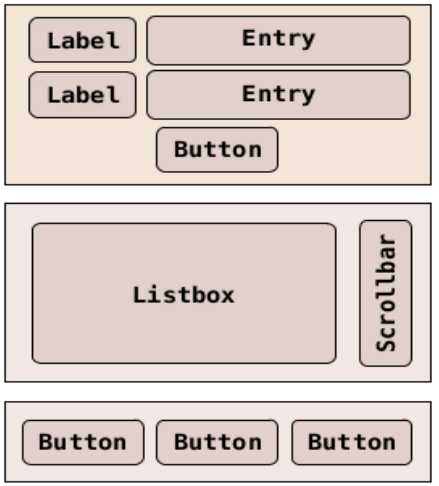
Code des couleurs

Oscillographe

Conception de l'application **tkPhone**



(a) Conception générale



(b) Détails des `Frame`

(c) L'application

La programmation graphique orientée objet

- Principe de la programmation pilotée par des événements.
- Notions de conception d'interface graphique.
- La bibliothèque standard `tkinter` :
  - une calculette simple ;
  - l'application `tkPhone`.

Chapitre 10/11

Programmation avancée

- Techniques procédurales
- Techniques objets
- Algorithmique

Pouvoir de l'introspection

C'est l'un des atouts de Python : la possibilité d'obtenir, à l'exécution, des informations sur les objets manipulés par le langage :

- L'aide en ligne. On a vu la fonction `help()` (ou `?` d'`ipython`)
- Les fonctions `type()`, `dir()` et `id()` fournissent respectivement le type, une liste des noms définis dans l'espace de noms et l'identification (unique) d'un objet
- Les fonctions `locals()` et `globals()` retournent les dictionnaires des noms locaux et globaux au moment de leur appel
- Le module `sys` fournit nombre d'informations générales concernant le système utilisé. Essayez dans ipython : `sys?`

Listes définies en compréhension

- Construction syntaxique se rapprochant de la notation utilisée en mathématiques :

$\{x^2 | x \in [2, 11[\}$ ⇔ `[x**2 for x in range(2, 11)]` ⇒ `[4, 9, 16, 25, 36, 49, 64, 81, 100]`

- Trois formes principales :

```
result1 = [x+1 for x in une_seq]
# A le même effet que :
result2 = []
for x in une_seq:
    result2.append(x+1)
```

```
result3 = [x+1 for x in une_seq if x > 23]
# A le même effet que :
result4 = []
for x in une_seq:
    if x > 23:
        result4.append(x+1)
```

```
result5 = [x+y for x in une_seq for y in une_autre]
# A le même effet que :
result6 = []
for x in une_seq:
    for y in une_autre:
        result6.append(x+y)
```

Exemples d'utilisations *pythoniques*

```
valeurs_s = ["12", "78", "671"]
# Conversion d'une liste de chaînes en liste d'entiers
valeurs_i = [int(i) for i in valeurs_s]     # [12, 78, 671]
```

```
# Initialisation d'une liste 2D
multi_liste = [[0, 0] for ligne in range(3)]
print(multi_liste)      # [[0, 0], [0, 0], [0, 0]]
```

Construction de noms de fichiers :

```
>>> [f"fic{n:03d}.txt" for n in [8, 16, 32, 64, 128, 512]]
['fic008.txt', 'fic016.txt', 'fic032.txt', 'fic064.txt', 'fic128.txt', 'fic512.txt']
```

Dictionnaires et ensembles définis en compréhension

- Dictionnaire :

```
>>> {k: v**2 for k, v in zip('abcde', range(1, 6))}
{'a': 1, 'b': 4, 'c': 9, 'd': 16, 'e': 25}
```

- Ensemble :

```
>>> {n for n in range(5)}
set([0, 1, 2, 3, 4])
```

Générateur

- C’est une **fonction** qui mémorise son état au moment de produire une valeur
- Elle génère une *exécution paresseuse* en calculant les valeurs demandées au fur et à mesure des besoins.
- Processus beaucoup plus efficace (en termes de mémoire) que le calcul, par exemple, d’une énorme liste en une seule fois
- La transmission d’une valeur produite s’effectue en utilisant le mot-clé `yield` à la place de `return`

Exemple de générateur

```
>>> def count_down(n) :
        print("Mise à feu :")
        while n > 0 :
            yield n
            n -= 1
>>> for val in count_down(5):
    print(val, end=" ")
Mise à feu :
5 4 3 2 1
```

Expression génératrice

Syntaxe presque identique à celle des listes en compréhension sauf qu'elle est entourée de parenthèses :

`(expression for i in s if condition)` est équivalent à :

```
for i in s :
    if condition:
        yield expression
```

- Par exemple, la liste en compréhension :

```
for i in [x**2 for x in range(1_000_000)]
```

génère la création d'un million de valeurs en mémoire **avant** de commencer la boucle.

- En revanche, dans l'expression génératrice :

```
for i in (x**2 for x in range(1_000_000))
```

la boucle commence immédiatement et les valeurs ne sont générées qu'**au fur et à mesure** des demandes.

Variable libre & cloture

- Une **variable libre** n'est ni locale ni globale, mais dans une fonction englobante
- Une fonction est une **cloture** lorsqu'elle référence une variable libre :

```
def plus_n(y) :
    def incremente(x) :
        return x + y
    return incremente
```

Utilisation de la cloture

```
>>> plus3 = plus_n(3)
```

La cloture `plus_n(3)` retourne la fonction `incremente` qui garde un lien vers la variable libre `y` (qui vaut `3`) :

```
>>> plus3(10)
>>> 13
>>> plus7 = plus_n(7)
>>> plus7(20)
>>> 27
```

Cloture

Notion de décorateur

permet de modifier le fonctionnement d'une fonction

Syntaxe :

```
@decorateur
def f():
    pass
…
…
f()
```

Notation **explicite**

Équivalent à :

```
def f():
    pass
…
f = decorateur(f)
…
…
f()
```

Point important

`f` n'est plus la fonction utilisée mais l'objet retourné par : decorateur(f)

Un décorateur est un **callable** (objet appelable comme une fonction) qui prend comme argument la fonction à décorer et retourne un callable :

```
f = decorateur(f)
```

- `decorateur(f)` retourne un objet : `obj`
- `f(a, b)` appelle en réalité : `obj(a, b)`

Temps d'exécution d'une fonction & Docstring d'une fonction décorée

Directive `lambda`

Permet de définir un objet **fonction anonyme** comportant **une expression** retournée par la fonction.

```
>>> # Retourne 's' si son argument est différent de 1, une chaîne vide sinon
>>> s = lambda x: "" if x == 1 else "s"
>>> s(1), s(3)
('', 's')
```

```
def polynome(a, b, c) :
    return lambda x : a*x**2 + b*x + c

p1 = polynome(3, -1, 4)
p2 = polynome(-1, 2, 0)
print(p1(1))   # 6
print(p1(2))   # 14
print(p2(10))  # -80
```

souvent utilisée dans les *callbacks* des interfaces graphiques

Plan

Programmation avancée

- Techniques procédurales
- Techniques objets
- Algorithmique

Accesseurs

- Le problème de l'encapsulation : en POO classique la visibilité de l'attribut d'un objet est **privée** par défaut (d'où le besoin des accesseurs). Or, en Python, tous les attributs d'un objet sont **publiques**

- Solutions :
  - faible : un nom d'attribut est préfixé par un caractère souligné, il est conventionnellement réservé à un usage interne (privé). Mais Python n'oblige à rien ! (W*e're all consenting adults here*)
  - meilleure : le principe de l'encapsulation est mis en œuvre par la notion de **propriété** (*property*)

Property

Une propriété (**`property`**) est un attribut d'instance possédant des fonctionnalités spéciales qui utilise la syntaxe des décorateurs.

```
class Temperature:
    def __init__(self):
        self.value = 0

    @property
    def celsius(self): # le nom de la méthode devient le nom de la propriété
        return self.value
    @celsius.setter
    def celsius(self, value): # le setter doit porter le même nom
        self.value = value

    @property
    def fahrenheit(self):
        return self.value * 1.8 + 32
    @fahrenheit.setter
    def fahrenheit(self, value):
        self.value = (value - 32) / 1.8
```

Property

Utilisation :

```
In [11]: t = Temperature()

In [12]: t.celsius = 37

In [13]: tc = t.celsius

In [14]: tf = t.fahrenheit

In [15]: tc, tf
Out[15]: (37, 98.60000000000001)
```

Duck typing

« S'il marche comme un canard et cancane comme un canard, alors c'est un canard ! »

Python ne s'intéresse qu'au comportement des objets :

même API => mêmes méthodes

```
>>> def calcule(a, b, c):
        return (a + b) * c

>>> calcule(1, 2, 3)
9
>>> calcule("Pommes ", "et oranges. ", 3)
'Pommes et oranges. Pommes et oranges. Pommes et oranges. '
```

Plan

Programmation avancée

- Techniques procédurales
- Techniques objets
- Algorithmique

Constructions algorithmiques de base

File : **FIFO** (*First In First Out*), on traite les éléments dans leur ordre d'arrivée

Pile : **LIFO** (*Last In First Out*), algorithme de la pile d'assiettes (Voir calculatrice RPN)

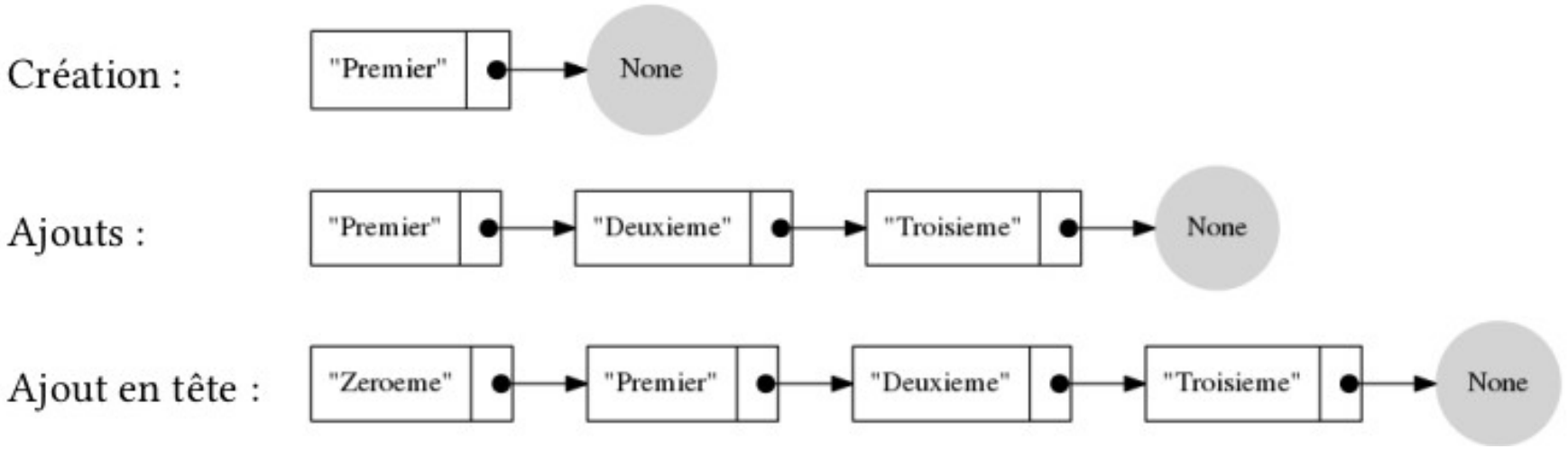
Autre variété : **FINO** (*First In Never Out*) : à éviter…

Liste chaînée

Arbre

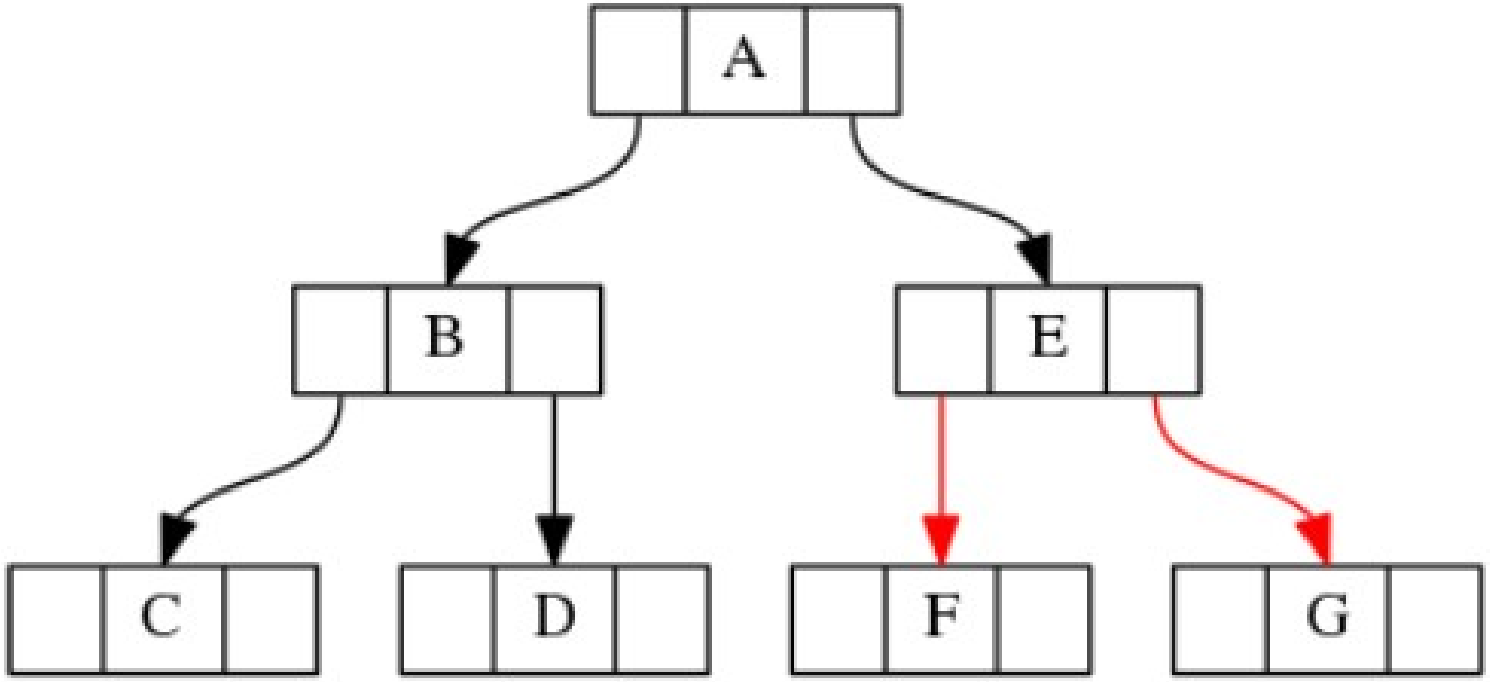
Listes chaînées

- Cette organisation de données est moins utilisée en Python, que dans les langages type C, Ada, C++...
- Elle est au cœur d'un des premiers langages informatiques, le LISP (1958)



Arbres

- Courants en algorithmique : organisation des fichiers, représentation interne de données triées, optimisation de certaines représentations…
- Les arbres informatique poussent la tête en bas !



Constructions algorithmiques de base

Fonctions récursives

Une fonction récursive comporte un appel à elle-même :

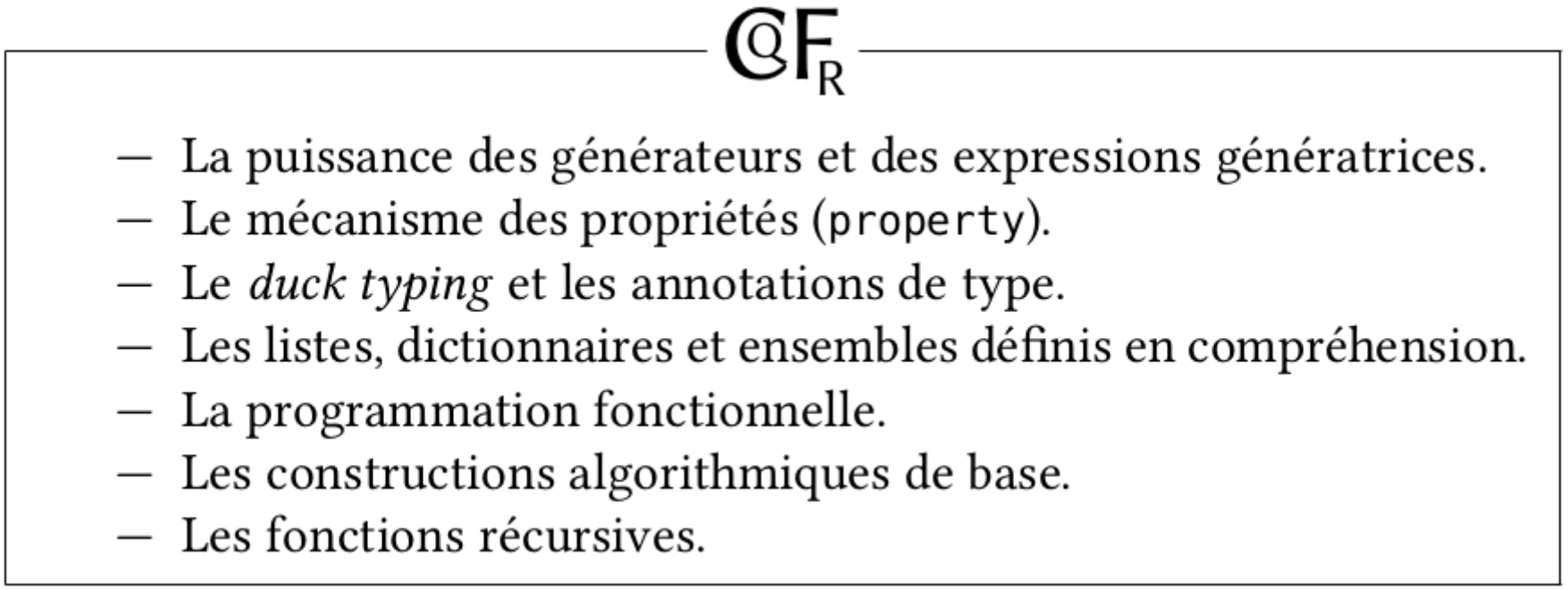
- besoin d'un cas de base sans récursion
- appels internes sur un cas « plus petit » pour terminer sur le cas de base

Voir exemple de l'algorithme de Horner : évaluation d'un polynôme de degré `n` en un point `x0`. Cette réécriture ne comporte plus que `n` multiplications

$$p(x_0) = ((\cdots((a_n x_0 + a_{n-1})x_0 + a_{n-2})x_0 + \cdots)x_0 + a_1)x_0 + a_0$$

Programmation avancée - exercices

Programmation avancée



Chapitre 11/11

Écosystème data science Python

- Numpy
- Pandas
- matplotlib

Batteries included

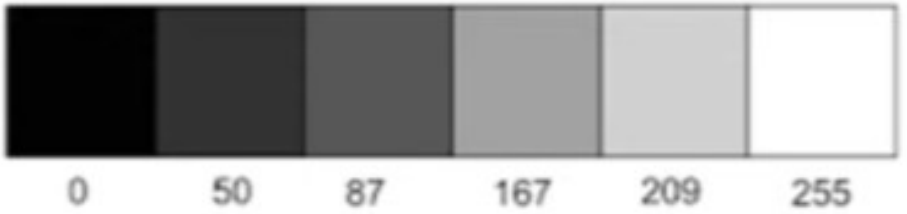
- On dit souvent que Python est livré « avec les piles » tant sa bibliothèque standard, riche de plus de 200 packages et modules, répond aux problèmes courants les plus variés
- Le *Python Package Index (https://pypi.org)* fournit une grande quantité de modules tierces
- Le domaine de la *Data Science* est en pleine extension. On va explorer ses outils de base

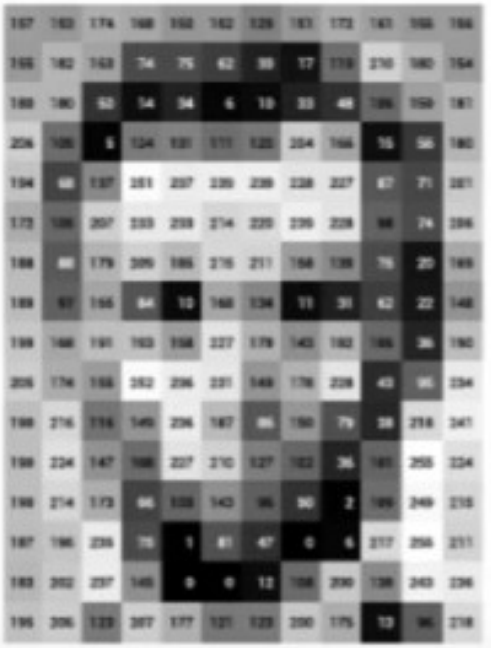
Numpy

- `numpy` est indispensable pour faire du calcul scientifique
- Pour modéliser les vecteurs, matrices et, plus généralement, les tableaux à n dimensions, `numpy` fournit les **ndarray** (type `np.array`)
- Deux différences majeures entre les tableaux `numpy` et les listes :
  - les tableaux sont **homogènes**, c'est-à-dire constitués d'éléments du même type. On trouvera donc des tableaux d'entiers, de flottants, de chaînes de caractères, etc.
  - **la taille des tableaux est fixée à la création**

La *souplesse* et la *simplicité* de Python, tant prônées par GvR, sont ici remplacées par *l'efficacité.*

Tableau 2D





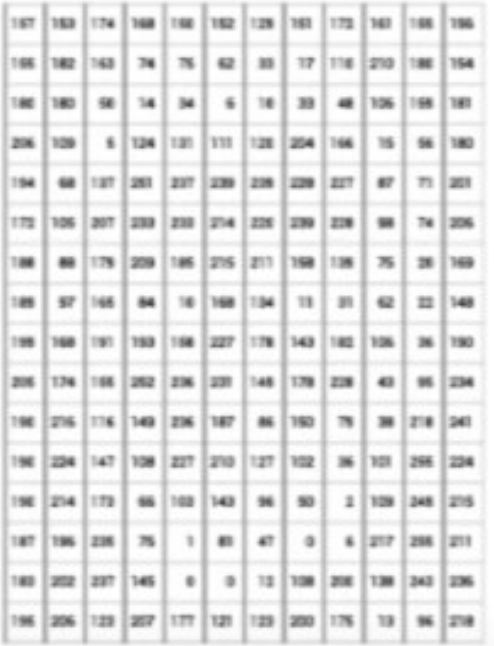
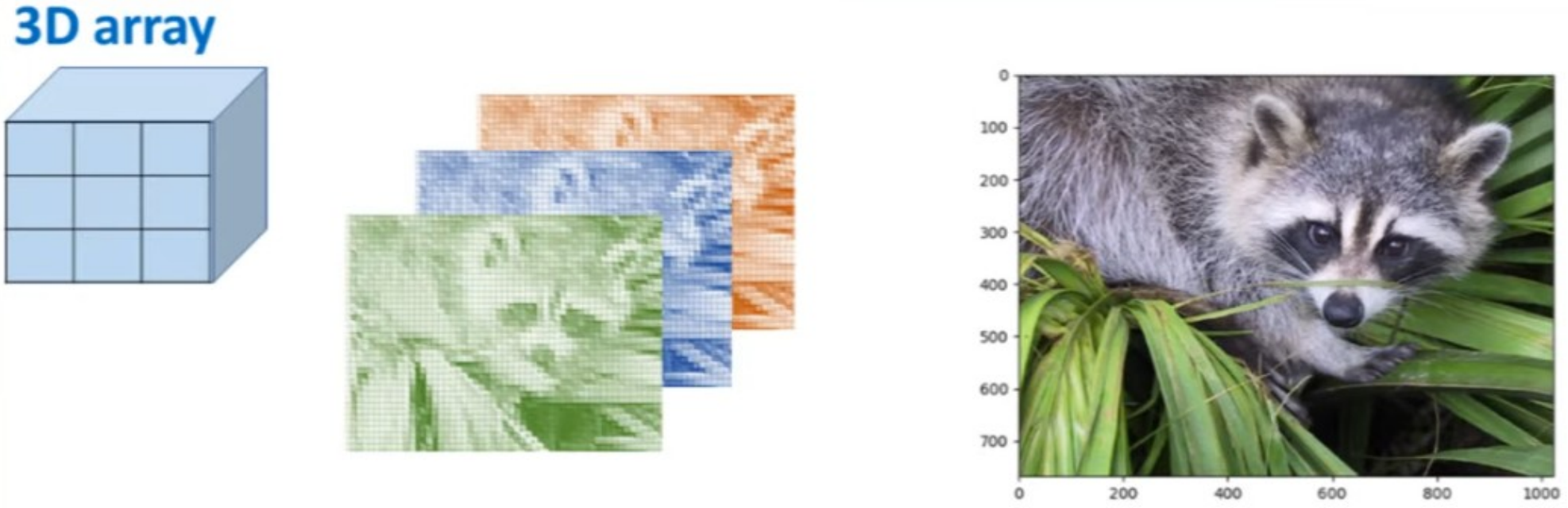


Tableau 3D



```
>>> import numpy as np                    # Convention de la communauté scientifique
>>> a = np.array(range(1, 5))             # Création à partir d'un itérateur
>>> a, a.dtype
(array([1, 2, 3, 4]), dtype('int64'))     # Type entier par défaut
>>> b = a * 2.5                           # Opération globale, vectorielle
>>> b, b.dtype                            # b est transtypé en flottant
(array([ 2.5,  5. ,  7.5, 10. ]), dtype('float64'))
>>> a @ b                                 # Multiplication matricielle
75.0
>>> c = np.array([5, 6, 7, 8])            # Création à partir d'une liste
>>> d = b + c                             # Addition globale, vectorielle
>>> d, d.dtype
(array([ 7.5, 11. , 14.5, 18. ]), dtype('float64'))
```

On crée un vecteur-ligne a d'entiers puis on le multiplie **globalement**, par le scalaire `2.5`

- En Python de base, on est passé de la boucle à la liste en compréhension puis à l'expression génératrice. Avec NumPy on passe à la vision globale, **vectorielle** : on applique une fonction à un tableau en utilisant une *Universal function*, souvent appelé *ufunc*. Toutes les opérations usuelles sont des *ufunc* NumPy.

- On créé souvent un `ndarray` en utilisant la méthode `arange()` quand on connaît le **pas**, ou la méthode `linspace()` quand on connaît le **nombre** :

```
>>> a, b = np.arange(5), np.arange(1.0, 2.0, 0.25)
>>> a, b
(array([0, 1, 2, 3, 4]), array([1.  , 1.25, 1.5 , 1.75]))
>>> c = np.linspace(0.0, 5.0, 6)
>>> c
array([0., 1., 2., 3., 4., 5.])
```

Forme d'un tableau NumPy

```
>>> m = np.array((range(11, 14), range(21, 24)))  # 2 lignes x 3 colonnes = 6
>>> m
array([[11, 12, 13],
       [21, 22, 23]])
>>> m.shape  # C'est un attribut, pas une fonction
(2, 3)
>>> m2 = m.reshape((3, 2))                        # 3 lignes x 2 colonnes = 6
>>> m2
array([[11, 12],
       [13, 21],
       [22, 23]])
```

Attention : m2 n'est pas une copie de m, c'est une **vue**.
Donc tout changement de m affecte m2

Attributs d'un `ndarray`

Attribut	*Signification*	*Exemple*
`shape`	tuple des dimensions	`(3, 5, 7)`
`ndim`	nombre de dimensions	`3`
`size`	nombre d'éléments	`3 * 5 * 7`
`dtype`	type des éléments	`np.float64`
`itemsize`	taille en octets d'un élément	`8`

Création de tableaux constants

```
>>> zeros = np.zeros(dtype=np.int8, shape=(2, 3))
>>> zeros
array([[0, 0, 0],
       [0, 0, 0]], dtype=int8)
>>> kvin = 5 * np.ones(shape=(3, 4))
>>> kvin
array([[5., 5., 5., 5.],
       [5., 5., 5., 5.],
       [5., 5., 5., 5.]])
```

Plan

Écosystème data science Python

- Numpy
- Pandas
- matplotlib

Plan

Écosystème data science Python

- Numpy
- Pandas
- matplotlib

Écosystème data science Python - exercices

Écosystème data science Python

- Richesse de la bibliothèque standard.
- Le domaine scientifique, un point fort de Python :
  - l'interpréteur de Python scientifique : `IPython` ;
  - la bibliothèque `NumPy` fournit le type de base `np.array` qui offre des opérations vectorielles très rapides ;
  - la bibliothèque `Pandas` permet la gestion des données avec les classes `Series` et `DataFrame` ;
  - la bibliothèque graphique `matplotlib`.
- `PyPI` et les bibliothèques tierces.
- La documentation et les tests des sources.

IHM morse ?

That's all Folks!