Algorithmique Programmation Objet

Python

Andrea G. B. Tettamanzi
Université de Nice Sophia Antipolis
Département Informatique
andrea.tettamanzi@unice.fr

CM - Séance 3

Introduction au langage Python

Plan

- Introduction générale au langage Python
- Présentation des éléments de base du langage
- La partie « orientée objet » du langage sera traitée dans la suite

Sources et Remerciments

- La première partie de cette présentation est basée sur la documentation officielle du langage, disponible sur le site www.python.org.
- La deuxième partie de cette présentation est une adaptation de l'excellent mémento sur les bases de Python 3 de Laurent Pointal.

Introduction

- Python est un langage de programmation
  - Multi-paradigme
  - Haut-niveau.
- Il favorise la programmation impérative structurée.
- Il supporte la programmation orientée objet.
- Il supporte la programmation fontionnelle.
- Il est doté de
  - typage dynamique fort,
  - gestion automatique de la mémoire par ramasse-miettes
  - système de gestion d'exceptions
- Il est, pour certains aspects, similaire à Perl, Ruby, Scheme, Smalltalk et Tcl.

Caractéristiques principales

- Langage Interprété
- Introspection
- Support intuitif pour la programmation orientée objet
- Modulaire, packages hierarchiques
- Gestion des erreurs basée sur les exceptions
- Types de données dynamiques haut-niveau
- Possède une riche bibliothèque standard
- Facilement extensible
- Documentation en ligne
- “Open”

Curiosités

- Créé en 1990 par Guido van Rossum
- Droits détenus par la Python Software Foundation
- Le langage est nommé après le groupe de comédiens anglais Monty Python
- Les versions successives à la 3.0 ont aboli la compatibilité descendante avec les versions 2.x

Bibliothèque standard

- Un point de force de python est la présence d'une grande bibliothèque standard (comme c'est le cas pour Java)
- Organisée hierarchiquement par modules
- Quelques modules :
  - os : interface avec le système d'exploitation
  - sys : accès à stdin, stdout, stderr, argv
  - math : fonctions mathématiques
  - random : générateur de nombres pseudo-aléatoires
  - urllib : accès au Web
  - ...

Types de base

entier, flottant, booléen, chaîne

```
  int  783    0    -192
float  9.23   0.0    -1.7e-6
 bool  True   False
  str  "Un\nDeux"    'L\'âme'
```

10^{-6}

retour à la ligne

' échappé

multiligne

```
"""X\tY\tZ
1\t2\t3"""
```

tabulation

non modifiable,
séquence ordonnée de caractères

Identifiants

Pour noms de variables, fonctions, modules, classes…

`a..zA..Z_` suivi de **`a..zA..Z_0..9`**
- accents possibles mais à éviter
- mots clés du langage interdits
- distinction casse min/MAJ

☺ **`a toto x7 y_max BigOne`**

☹ ~~**`8y and`**~~

Affectation de variables

```
x = 1.2+8+sin(0)
```

valeur ou expression de calcul

nom de variable (identificateur)

```
y,z,r = 9.2,-7.6,"bad"
```

noms de variables

conteneur de plusieurs valeurs (ici un tuple)

`x+=3` ← incrémentation

décrémentation → `x-=2`

`x=None` valeur constante « non défini »

Types conteneurs

- séquences ordonnées, accès index rapide, valeurs répétables

```
  list   [1,5,9]    ["x",11,8.9]   ["mot"]      []
 tuple   (1,5,9)     11,"y",7.4    ("mot",)     ()
```

non modifiable

expression juste avec des virgules

`str` en tant que séquence ordonnée de caractères

- sans ordre *a priori*, clé unique, accès par clé rapide ; clés = types de base ou tuples

```
  dict   {"clé":"valeur"}                        {}
```

dictionnaire
couples clé/valeur

```
         {1:"un",3:"trois",2:"deux",3.14:"π"}
```

ensemble

```
   set   {"clé1","clé2"}    {1,9,3,0}     set()
```

Conversions

`type(`*expression*`)`

`int("15")` on peut spécifier la base du nombre entier en 2ème paramètre

`int(15.56)` troncature de la partie décimale (`round(15.56)` pour entier arrondi)

`float("-11.24e8")`

`str(78.3)` et pour avoir la représentation littérale → `repr("Texte")`

voir aussi le formatage de chaînes, qui permet un contrôle fin

`bool` → utiliser des comparateurs (avec `==`, `!=`, `<`, `>`, …), résultat logique booléen

`list("abc")` → *utilise chaque élément de la séquence en paramètre* → `['a','b','c']`

Conversions

```
dict([(3,"trois"),(1,"un")])
```
→ `{1:'un',3:'trois'}`

```
set(["un","deux"])
```
→ *utilise chaque élément de la séquence en paramètre* → `{'un','deux'}`

```
":".join(['toto','12','pswd'])
```
→ `'toto:12:pswd'`

chaîne de jointure — séquence de chaînes

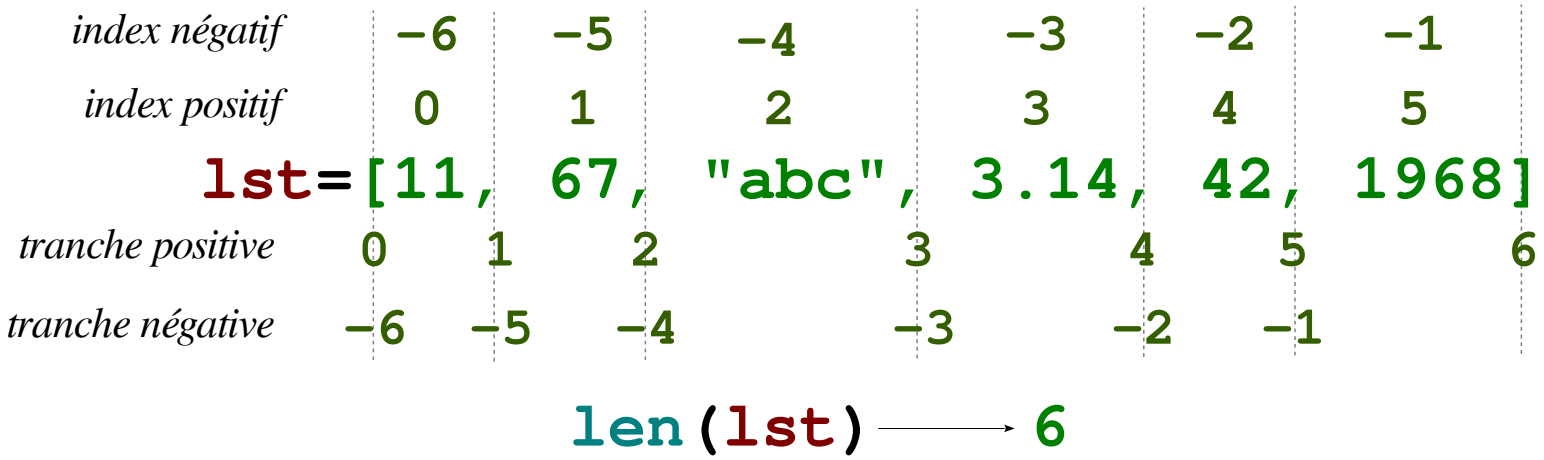
```
"des mots  espacés".split()
```
→ `['des','mots','espacés']`

```
"1,4,8,2".split(",")
```
→ `['1','4','8','2']`

chaîne de séparation

Indexation des séquences

Valable pour les listes, tuples, chaînes de caractères, …




```
len(lst) ⟶ 6
```

accès individuel aux éléments par [*index*]

```
lst[1]→67
lst[-2]→42
lst[0]→11    le premier
lst[-1]→1968 le dernier
```

Indexation des séquences

```
lst=[11, 67, "abc", 3.14, 42, 1968]
```

Accès à des sous-séquences par [*tranche début*:*tranche fin*:*pas*]

```
lst[:-1]→[11,67,"abc",3.14,42]
lst[1:-1]→[67,"abc",3.14,42]
lst[::2]→[11,"abc",42]
lst[:]→[11,67,"abc",3.14,42,1968]
lst[1:3]→[67,"abc"]      lst[-3:-1]→[3.14,42]
lst[:3]→[11,67,"abc"]    lst[4:]→[42,1968]
```

Indication de tranche manquante → à partir du début / jusqu'à la fin.

Sur les séquences modifiables, utilisable pour suppression `del lst[3:5]`
et modification par affectation `lst[1:4]=['hop',9]`

Logique booléenne

Comparateurs: `<` `>` `<=` `>=` `==` `!=`
≤ ≥ = ≠

`a and b` conjontion logique
les deux en même temps

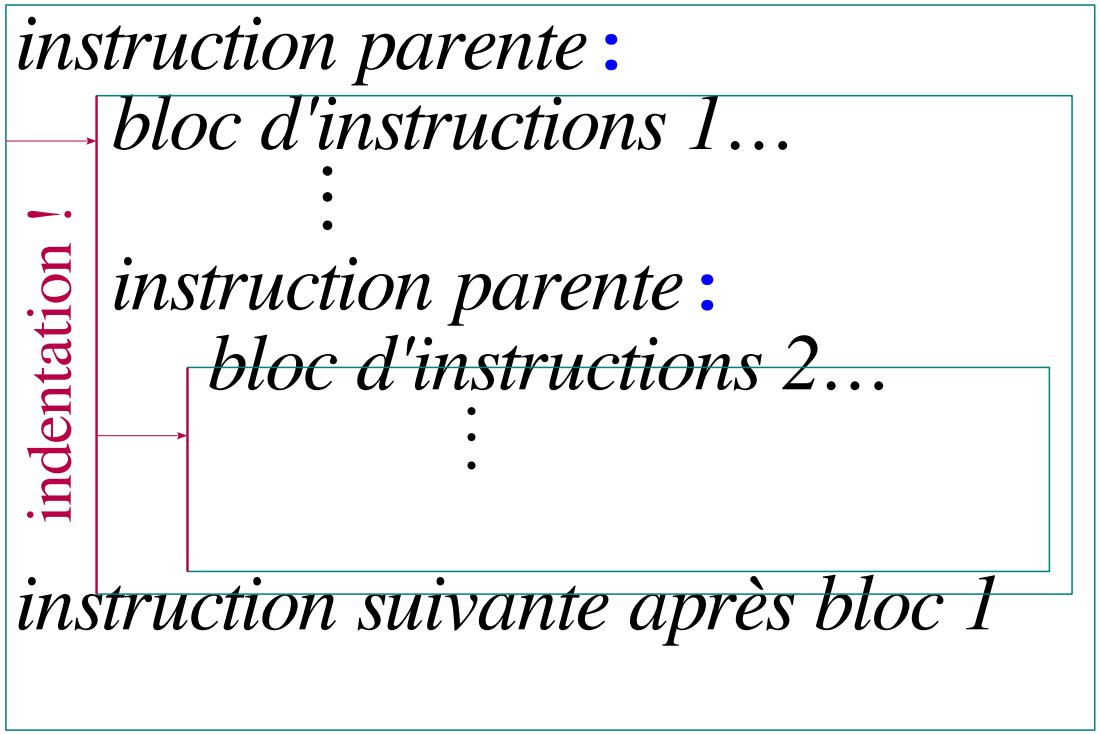
`a or b` disjonction logique
l'un ou l'autre ou les deux

`not a` négation logique

`True` valeur constante vrai

`False` valeur constante faux

Blocs d'instructions



Instruction conditionnelle

`if` *expression logique* **:**
    bloc d'instructions

bloc d'instructions exécuté uniquement si une condition est vraie

Combinable avec des sinon si, sinon si... et un seul sinon final, exemple :

```
if x==42:
    # bloc si expression logique x==42 vraie
    print("vérité vraie")
elif x>0:
    # bloc sinon si expression logique x>0 vraie
    print("positivons")
elif bTermine:
    # bloc sinon si variable booléenne bTermine vraie
    print("ah, c'est fini")
else:
    # bloc sinon des autres cas restants
    print("ça veut pas")
```

Maths

☝ *nombres flottants… valeurs approchées !*

Opérateurs: `+ - * / // % **`

× ÷ ↑ ↑ a^b

÷ entière reste ÷

```
(1+5.3)*2→12.6
abs(-3.2)→3.2
round(3.57,1)→3.6
```

angles en radians

```
from math import sin,pi…
sin(pi/4)→0.707…
cos(2*pi/3)→-0.4999…
acos(0.5)→1.0471…
sqrt(81)→9.0      √
log(e**2)→2.0     etc. (cf doc)
```

Instruction boucle conditionnelle

Bloc d'instructions exécuté tant que la condition est vraie

`while` *expression logique* **:**
→ *bloc d'instructions*

```
s = 0
i = 1
```
*initialisations **avant** la boucle*

*condition avec au moins une valeur variable (ici **`i`**)*

```
while i <= 100:
    # bloc exécuté tant que i ≤ 100
    s = s + i**2
    i = i + 1
```
☝ *faire varier la variable de condition !*

$$s = \sum_{i=1}^{i=100} i^2$$

```
print("somme:",s)
```
résultat de calcul après la boucle

☝ *attention aux boucles sans fin !*

Contrôle de boucle

`break` *sortie immédiate*

`continue` *itération suivante*

Instruction boucle itérative

bloc d'instructions exécuté pour chaque élément d'une séquence de valeurs ou d'un itérateur

```
for variable in séquence:
    bloc d'instructions
```

Parcours des **valeurs** de la séquence

```
s = "Du texte"
cpt = 0
```

*initialisations **avant** la boucle*

variable de boucle, valeur gérée par l'instruction **`for`**

```
for c in s:
    if c == "e":
        cpt = cpt + 1
print("trouvé",cpt,"'e'")
```

Comptage du nombre de **`e`** *dans la chaîne.*

boucle sur dict/set = boucle sur séquence des clés

utilisation des tranches pour parcourir un sous-ensemble de la séquence

Instruction boucle itérative

Parcours des **index** de la séquence
- changement de l'élément à la position
- accès aux éléments autour de la position (avant/après)

```
lst = [11,18,9,12,23,4,17]
perdu = []
for idx in range(len(lst)):
    val = lst[idx]
    if val> 15:
        perdu.append(val)
        lst[idx] = 15
print("modif:",lst,"-modif:",perdu)
```

Bornage des valeurs supérieures à 15, mémorisation des valeurs perdues.

Génération de séquences d'entiers

Très utilisée pour les boucles itératives for

par défaut 0 — non compris

`range``(`*[début,] fin [,pas]*`)`

```
range(5)      → 0 1 2 3 4
range(3,8)    → 3 4 5 6 7
range(2,12,3) →  2 5 8 11
```

`range` retourne un « générateur », faire une conversion en liste pour voir les valeurs, par exemple:

```
print(list(range(4)))
```

Opérations sur conteneurs

`len(c)` → nb d'éléments

`min(c)` `max(c)` `sum(c)`

`sorted(c)` → *copie* triée

*Note: Pour dictionnaires et ensembles, ces opérations travaillent sur les **clés**.*

```
for idx,val in enumerate(c):
    bloc d'instructions
```

Boucle directe sur index et valeur en même temps

`val in c` → booléen, opérateur `in` de test de présence (`not in` d'absence)

*Spécifique aux **conteneurs de séquences** (listes, tuples, chaînes) :*

`reversed(c)` → *itérateur* inversé `c*5` → duplication `c+c2` → concaténation

`c.index(val)` → position `c.count(val)` → nb d'occurences

Opérations sur listes

☝ modification de la liste originale

`lst.append(item)` ajout d'un élément à la fin

`lst.extend(seq)` ajout d'une séquence d'éléments à la fin

`lst.insert(idx,val)` insertion à une position

`lst.remove(val)` suppression d'un élément à partir de sa valeur

`lst.pop(idx)` suppression de l'élément à une position et retour de la valeur

`lst.sort()` `lst.reverse()`

tri / inversion de la liste *sur place*

Opération sur dictionnaires

`d[`*clé*`]=`*valeur*
`d[`*clé*`]`→*valeur*

`d.clear()`
`del d[`*clé*`]`

`d.update(d2)` — *mise à jour/ajout des couples*

`d.keys()`
`d.values()`
`d.items()`
— *vues sur les clés, valeurs, couples*

`d.pop(`*clé*`)`

Opérations sur ensembles

Opérateurs:

`|` → union (caractère barre verticale)

`&` → intersection

`-` `^` → différence/diff symétrique

`<` `<=` `>` `>=` → relations d'inclusion

```
s.update(s2)
s.add(clé)  s.remove(clé)
s.discard(clé)
```

Complexité des opérations sur conteneurs

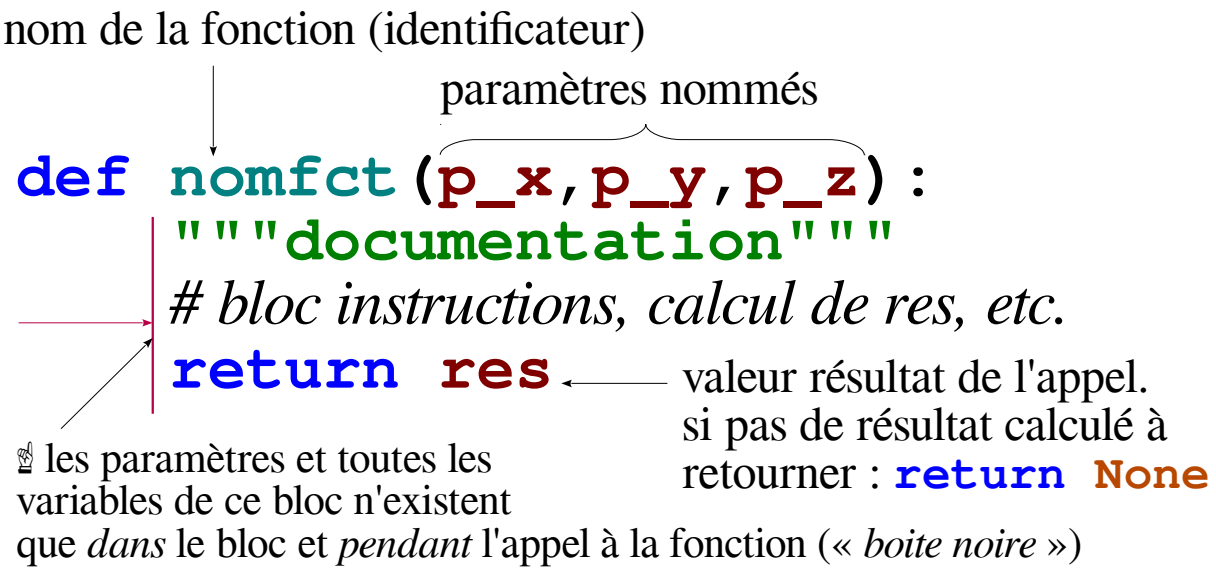
- Liste
  - Append : O(1)
  - Insert : O(n)
  - Get/Set item : O(1)
  - Delete item : O(n)
  - Itération : O(n)
  - Get tranche : O(k)
  - Delete tranche : O(n)
  - Set tranche : O(n + k)
  - Min, max, x in s : O(n)
  - Tri : O(n log n)

- Ensemble
  - x in S : O(n)
  - Union : O(n + m)
  - Intersection : O(nm)
  - Différence : O(n)
  - Diff. Symmetrique : O(mn)

- Dictionnaire
  - Get/Set item : O(n)
  - Delete item : O(n)
  - Itération : O(n)

Source : http://wiki.python.org/moin/TimeComplexity

Définition de fonction



Appel de fonction

```
r = nomfct(3,i+2,2*i)
```

un argument par paramètre

récupération du résultat retourné (si nécessaire)

Arguments

- Normalement, les arguments sont positionnels
  - Il sont reconnus par leur position dans la défintion d'une fonction
- D'autres types d'arguments sont disponibles.
- Arguments par mots clés (keyword arguments) :
  - Syntaxe : mot_clé=valeur
  - Ils doivent toujours suivre les arguments positionnels
- Arguments variadiques :
  - Syntaxe : *arg
  - Prend un tuple contenant tous les arguments passés à une fonction
  - Syntaxe : **arg → même chose, mais dictionnaire

Fichiers

```
f = open("fic.txt","w",encoding="utf8")
```

variable fichier pour les opérations

nom du fichier sur le disque (+chemin…)

mode d'ouverture
- `'r'` lecture (read)
- `'w'` écriture (write)
- `'a'` ajout (append)…

encodage des caractères pour les fichiers textes:
`utf8` `ascii` `latin1` …

cf fonctions des modules `os` et `os.path`

Écriture et Lecture

en **écriture**

```
f.write("coucou")
```

☝ *fichier texte → lecture / écriture de **chaînes** uniquement, convertir de/vers le type désiré*

en **lecture**

chaîne vide si fin de fichier

```
s = f.read(4)
```

si nb de caractères pas précisé, lit tout le fichier

lecture ligne suivante

```
s = f.readline()
```

```
f.close()
```

☝ ne pas oublier de refermer le fichier après son utilisation !
Fermeture automatique Pythonesque : `with f as open(…):`

très courant : boucle itérative de lecture des lignes d'un fichier texte :

```
for ligne in f :
    bloc de traitement de la ligne
```

Formatage de chaînes

directives de formatage — valeurs à formater

```
"modele{} {} {}".format(x,y,r) ⟶ str
"{sélection:formatage!conversion}"
```

- **Sélection** :
  - `2`
  - `x`
  - `0.nom`
  - `4[clé]`
  - `0[2]`

Exemples :

```
"{:+2.3f}".format(45.7273)
→'+45.727'
"{1:>10s}".format(8,"toto")
→'      toto'
"{!r}".format("L'ame")
→'"L\'ame"'
```

- **Formatage** :

car-rempl. *alignement* *signe* *larg.mini*`.`*précision~larg.max* *type*

`< > ^ =` `+ -` *espace* `0` au début pour remplissage avec des 0

entiers: `b` binaire, `c` caractère, `d` décimal (défaut), `o` octal, `x` ou `X` hexa…
flottant: `e` ou `E` exponentielle, `f` ou `F` point fixe, `g` ou `G` approprié (défaut),
`%` pourcentage
chaîne : `s` …

- **Conversion** : `s` (texte lisible) ou `r` (représentation littérale)

Merci de votre attention