Langages Dynamiques

Notes de cours

Sylvain Conchon

Sylvain.Conchon@lri.fr

03 décembre 2019

Le langage Python3 : une brève introduction

Historique

Langage inventé par Guido Van Rossum

- Première release en 1991
- Python 1.0 en 1994
- Python 2.0 en 2000
- Python 3.0 en 2008

Développé initialement comme un langage de script (interpréteur de commandes) pour Amoeba (OS)

Les types prédéfinis

- Les types élémentaires et opérations de base : `string`, `int`, `float`, `bool`
- Les types construits : tuples, listes, set, dictionnaires

Les chaînes de caractères (1/2)

Les chaînes se définissent entre deux apostrophes (') ou deux guillemets (").

```
print('Hello world')
print("j'adore Python")
```

On peut accéder aux éléments d'une chaîne et à sa longueur.

```
s1 = 'Hello'
print(s1[2])  # on commence à 0
print(s1[-1]) # dernier élément de la chaîne
print(len(s2)) # len s'applique à d'autres valeurs
```

Opérations arithmétiques sur les chaînes.

```
s2 = ' world'
print(s1+s2)
s3 = "abc" * 3
print(s3)
```


Les chaînes de caractères (2/2)

Les chaînes ne sont pas modifiables.

```
s1[0] = 'e' # --> Error : immutable strings
```

Extraction de sous-chaînes.

```
s4 = 'bonjour'
print(s4[0:2], s4[0:-2], s4[0:6:2], s4[::3])
```

Test d'appartenance

```
'onj' in s4       # True
```

Égalités (structurelle et physique) sur les listes

```
s4 == s4[:]       # True
s4 is s4[:]       # True pour les string
s4[0:2] == 'bo'   # True
s4[0:2] is 'bo'   # False
```


Int et Float

Les entiers et flottants et leurs opérateurs.

```
12 + 4.5 * (6 % 2) + (5 // 2) - (62 / 10) + (8.2 // 2)

y = 100_000 + 5_000_000 # lisibilité
print(62 / 10)
print(5 // 2)           # division entière
print(8.2 // 2)         # marche aussi sur les flottants
```


Booléens

Les booléens True et False et leurs opérateurs.

```
True and False or (not (True and False))

print (1 <= 2 and 4 > 3 and 4 == 4)
print (5 != 67 and  'tu' is not 're')
```

La construction `if-then-else` dans les expressions

```
print (4 if f(x) else 5)
```

Tuples

Définition d'un tuple

```
c1 = (1, 2, 'abc')
```

Accès aux éléments d'un tuple

```
print (c1[0] + c1[1])
c2 = (3,(4,'toto',True),'ok')
```

Les tuples ne sont pas modifiables

```
c2[0] = 5 # --> Error : tuples are immutables
```

Filtrage d'un tuple

```
(x,y) = (4, ('titi', 4.5, False))
print (y[1] + 4)
```


Les listes (1/4)

Définitions explicites des listes.

```
l1 = [1,2,3]
```

Les listes peuvent être hétérogènes (typage dynamique).

```
l2 = [1, 2.4, 'bonjour', [3,'ab'], [], 8]
```

Accès aux éléments d'une liste

```
print(l[0], l[-1])
```

Listes de listes (de listes, ...)

```
t = [ [1,1,1] , [2,2,2] , [3,3,3] ]
print(t[0], t[0][2])
```


Les listes (2/4)

Définitions par compréhension

```
l8 = [1, 2, 3, 4, 5, 6, 7, 8 ,9]
[ x * 2 for x in l8]
[x * 2 for x in range(1,10)]
list(range(0,10))
[ (i,j) for i in range(0,4) for j in range(0,3)]
[ (i,j) for i in range(0,6) for j in range(0,6) if i < j]
```


Les listes (3/4)

Longueur d'une liste `l` : `len(l)`

Extraction de sous-listes

```
l1 = [1, 2, 3, 4, 5, 6]
l1[1:4]
l3 = l1[:]
print(l1 == l3)
print(l1 is l3)
```

Opérateur `in` sur les listes : `3 in l1`

Concaténation de listes

```
l6 = l1 + l2
l7 = l6 + [10]
```

Les listes (4/4)

Duplication de listes

```
l4 = [1,2,3] * 4
l4[0] = 10
print(l4)
```

Attention, duplication seulement au premier niveau !

```
l5 = [[1,2,3]] * 2
l5[0][0] = 10
print(l5)
```


Complexités

Opération	Exemple	Classe
Index	l[i]	O(1)
Store	l[i] = 0	O(1)
Length	len(l)	O(1)
Append	l.append(5)	O(1)
Pop	l.pop()	O(1)
Slice	l[a:b]	O(b-a)
Extend	l.extend(...)	O(len(...))
Construction	list(...)	O(len(...))
check ==, !=	l1 == l2	O(N)
Insert	l[a:b] = ...	O(N)
Delete	del l[i]	O(N)
Containment	x in/not in l	O(N)
Copy	l.copy()	O(N)
Remove	l.remove(...)	O(N)
Popi	l.pop(i)	O(N)
Reverse	l.reverse()	O(N)
Multiply	k*l	O(k N)

Les ensembles

Définitions d'ensembles

```
s1 = { 1,2,1,4,10 }
s2 = set('bonjour')
```

Les éléments d'un ensemble doivent être non modifiables

```
set([1],[2]) # Error
```

Opérations ensemblistes impératives

```
s1.add('t')
s1.remove('a')
```

Opérations ensemblistes fonctionnelles

```
s3 = s1.union(s2)
s4 = s1.difference(s2)
s5 = s1.intersection(s4)
```

Cardinalité et appartenance : `len(s1)`, `'a' in s1`

Les dictionnaires

Initialisation d'un dictionnaire

```
d = { 'toto' : 'rr', 'titi' : 42, (5,6) : 25 }
```

Accès aux définitions

```
print (d['toto'], d['titi'], d[(5,6)])
# d['r'] -> KeyError 'r'
```

Suppression d'une clé :

```
del d['toto']
# del d['uu'] -> KeyError : 'uu'
x = d.pop('titi')
```

Test d'appartenance : `'titi' in d`

Les boucles

- Boucle `for`
- Boucle `While`

Les boucles `for`

On peut utiliser des boucles `for` sur tous les objets itérables (tuples, listes, dictionnaires, strings, etc.)

```
for i in range(0,10):
   print(i,' ',end="")

for i in range(9,-1,-1):
   print(i,' ',end="")
```


Les boucles `while`

```
t = [0,0,0,39,42,5]
i = 0
while i < len(t) and t[i] == 0:
  i+=1
print('i:',i)
```


Les fonctions

- Définitions de fonctions
- Lambdas
- Itérateurs

Les fonctions

Définition d'une fonction :

```
def f(x,y):
   return x+y
```

Appel d'une fonction :

```
print('Resultat:', f(5,4))
```

Les fonctions sont implicitement récursives :

```
def fact(n):
   if n == 0:
       return 1
   else:
       return n * fact(n-1)
```


Fonctions anonymes

Python permet de définir des fonctions anonymes sous la forme d'*expressions* `lambda`.

L'affectation suivante

```
f = (lambda x,y : x * (y + y))
```

définit une fonction `f` qui prend deux arguments `x` et `y` et qui renvoie `x * (y + y)`.

```
print('Resultat:', f(5,4))
```


Itérateurs

map

```
list(map(f,[e1, e2, ..., en])) == [f(e1), f(e2), ..., f(en)]
```

reduce

```
reduce(f,[e1, e2, ..., en],v) == f(...(f(f(v,e1),e2), ...),en)
reduce(f,[e1, e2, ..., en]) == f(...(f(f(e1,e2),e3), ...),en)
```

Attention, ajouter la directive suivante pour utiliser `reduce` :

```
from functools import reduce
```

filter

```
filter(f,l) == [ v for v in l if f(v)]
```


La portée des variables (1/3)

En Python, les variables sont locales, a priori...

```
def h1():
    loc_x = 10     # loc_x est affectée, donc locale
    return loc_x

print('h1:',h1())
print(loc_x)       # NameError: name 'loc_x' is not defined
```

Si une variable n'est pas définie localement, alors elle est recherchée dans l'environnement des variables globales

```
def h2():
    print(s)       # s est globale ici
    return

s = 'bonjour'
h2()
```

Portée des variables (2/3)

Pour être sûr d'avoir bien tout compris...

Que vaut `z` après l'appel à `h3` ?

```
z = 100
def h3():
   z = 45
   return
h3()
```

La définition suivante est refusée. Pourquoi ?

```
def h4():
   z = z + 3
   return
```


Portée des variables (3/3)

Quels sont les statuts de `y` et `z` dans `h5` ?

```
def h5():
   y = z + 10
   return y
```

Même question pour la variable `h` dans `h6` ?

```
h = 1000
def h6(b):
   if b:
       y = h + 42
   else:
       h = 200
```


Exos sur les listes

En utilisant les itérateurs `map` et `reduce`, écrire les fonctions suivantes :

`somme(l)` qui renvoie la somme des éléments d'une liste

`maximum(l)` qui renvoie le plus grand élément d'une liste

`fst(l)` qui renvoie une liste constituée des éléments gauche d'une liste de paires

`permute(l)` qui remplace les `0` par des `1` (et réciproquement) dans une liste `l` (constituée uniquement de `0` et de `1`)

`compte0(l)` qui compte le nombre de `0` dans la liste `l`

`pgs(v,l)` qui renvoie la longueur de la plus grande séquence de `v` dans la liste `l`

`dernier_pair(l)` qui renvoie le dernier élément pair de la liste `l` (et `None` s'il n'existe pas)

`second(l)` qui renvoie le deuxième plus grand élément d'une liste (supposée non vide)

Les modules

Les modules

Un module Python est simplement un fichier qui contient des définitions et des instructions.

La commande suivante ajoute le nom du module `module1` dans l'environnement des variables globales.

```
import module1
```

Il est ensuite possible d'utiliser les fonctions ou valeurs définies dans ce fichier.

```
import module1
y = module1.x + module1.f(100)
print(y)
```


Effets de la commande `import`

Les instructions du module importées sont exécutées au moment de l'import. Par exemple, dans l'exemple ci-dessus, si `module1.py` contient :

```
x = 42
print(x)
def f(x):
   return x+1
```

Alors, la valeur 42 sera affichée, puis `y` sera calculée et enfin l'instruction `print(y)` sera exécutée.

Import de modules (suite)

Il est possible d'importer une sélection de valeurs (fonctions) dans un module.

On peut également donner un autre nom au module importé.

Enfin, les noms des valeurs importées par un module cachent les valeurs de même nom importées précédemment.

```
from module2 import f1, f2
import module3 as m
from module4 import f2 # cache la fonction f2 de module2
```


Les objets

La couche objet

Le premier concept des langages à objets est celui de classe qui permet de regrouper dans un même « espace » données et algorithmes sur ces données.

```
from math import sqrt

class Vecteur:
    def __init__(self):
        self.x = 0
        self.y = 0

    def norme(self):
        return sqrt(self.x * self.x + self.y * self.y)
```


Objets

Les objets sont des *instances* particulières d'une classe.

Par exemple,

```
v = Vecteur()
```

a pour effet de déclarer une nouvelle variable `v` dont la valeur est une nouvelle instance de la classe `Vecteur`

L'objet est alloué sur le tas.

Champs d'un objets

Les données contenues dans un objets sont stockées dans les champs (on dit également attributs ou variables d'instances) de l'objet.

Dans la classe `Vecteur`, les champs sont `x` et `y`.

On peut accéder aux champs de `v`, et les modifier, avec la notation pointée

```
print("(",v.x," , ",v.y,")")
v.x = 1
v.y = 1
```


Constructeurs

La fonction `__init__` est particulière, c'est le constructeur de la classe.

Le rôle d'un constructeur est d'initialiser les champs de l'objet qu'il prend en argument (paramètre self). Il est appelé automatiquement quand on déclare un nouvel objet.

Le constructeur de classe peut également recevoir des paramètres. Cela est utile pour initialiser les variables d'instances.

```
class Vecteur:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Ainsi, on peut créer un nouvel objet de la manière suivante :

```
v = Vecteur(10,8)
```

Encapsulation

Un des intérêts de la programmation objet est l'encapsulation, c'est-à-dire la possibilité d'interdire l'accès à certains champs d'un objet en les rendant invisibles à l'extérieur de la classe.

Cela permet par exemple de maintenir des invariants (par exemple que l'on manipule toujours des vecteurs normalisés).

Il n'y a pas vraiment de mécanisme pour déclarer des champs privés en Python, mais on peut *simuler* cela en ajoutant `__` (deux caractères soulignés) devant le nom du champ.

```
class Vecteur:
    def __init__(self, x, y):
        self.__x = x
        self.__y = y
```

Ces champs seront toujours accessibles à l'extérieur, mais leur nom sont automatiquement transformés en `_Vecteur__x`.

Méthodes

Les fonctions définies dans une classe sont appelées méthodes.

Une méthode définie dans une classe s'applique sur un objet de la même classe de la manière suivante :

```
v.norm()
```

Comme pour le constructeur de classe, une méthode prend comme premier argument l'objet sur lequel elle est appliquée. Par convention, cet argument doit être nommé self.

```
def norme(self):
    x2 = self.__x * self.__x
    y2 = self.__y * self.__y
    return sqrt(x2 + y2)
```


Variables statiques (ou de classe)

Une classe peut aussi contenir ses propres variables, appelées variables de classe. Ces variables sont liées à la classe et non aux instances de la classe.

Exemple :

```
class A:
    v = 5
    def __init__(self,n):
        self.x = A.v + n
```

La variable `v` appartient à la classe `A` et on y accède en écrivant `A.v`

Ces variables sont également modifiables.

```
A.v = 10
p2 = A(6)
print(p2.x)
```


Méthodes statiques (ou de classe)

Il est également possible de définir des méthodes appartenant à une classe. Il s'agit de méthodes statiques (ou de classe).

En Python, il suffit de définir une méthode sans ajouter l'argument self et de précéder la définition par `@staticmethod` comme ceci :

```
class B:

    @staticmethod
    def g(x):
        return x + 1

print (B.g(5))
```

Héritage

Un autre concept important de la programmation Objet est celui d'héritage : une classe peut être définie comme héritant d'une ou plusieurs autres classes.

Les objets de la classe définie par héritage héritent de tous les champs et méthodes des classes héritées, auxquels ils peuvent ajouter de nouveaux champs ou nouvelles méthodes.

Héritage simple

Certains langage de programmation ne permettent à une classe d'hériter que d'une seule classe, c'est l'héritage simple.

```
class A:
    v = 10
    def __init__(self,x):
        self.x = A.v + x

    def f(self,y):
        return self.x - y

class B(A):
    def g(self,z):
        return z - self.x

p = B(7)
print (p.g(100) + p.f(10))
```


Héritage simple et initialisation

L'initialisation des objets d'une classe définie par héritage se fait par un appel explicite au constructeur de la classe mère (ou super classe) à l'aide de la notation `super()`.

```
class B(A):
  def __init__(self,w):
      self.w = w
      super().__init__(w+5)
```

L'appel au constructeur de la classe mère peut aussi se faire de la manière suivante :

```
class B(A):
 def __init__(self,w):
     self.w = w
     A.__init__(self,w+5)
```


Héritage multiple (1/1)

Une classe peut également hériter de plusieurs classes. C'est l'héritage multiple.

```
class A:
    def __init__(self,n): self.age = n
    def incr(self): self.age += 1

class B:
    def __init__(self,n): self.nom = n
    def affiche(self): print(self.nom)

class C(A,B):
    def __init__(self,p,a):
        B.__init__(self,p)
        A.__init__(self,a)

    def anniversaire(self):
        self.incr()
        print("C'est l'anniversaire de "); self.affiche()
        print("Il a ", self.age, "ans")

p = C("Toto",10)
p.anniversaire()
```

Héritage multiple (2/2)

Il convient de faire attention quand une classe hérite de plusieurs autres classes qu'il n'y ait pas conflits entre les noms de variables d'instances ou de méthodes.

Python n'aide pas beaucoup dans ce cas : aucun messages d'erreur et on ne sait pas quelles variables ou méthodes seront utilisées :-(

Exemple

```
class Graphical:
    def __init__(self,x=0,y=0,w=0,h=0):
        self.x = x; self.y = y;
        self.width = w; self.heigth = h
    def move(self,dx,dy): self.x += dx; self.y +=dy
    def draw(self):
        # fonction qui ne fait rien
        return

class Rectangle(Graphical):
    def __init__(self,x1,y1,x2,y2):
        super().__init__((x1+x2)/2, (y1+y2)/2, abs(x1-x2), abs(y1-y2))
    def draw(self):
        print("je dessine un rectangle")
        return

r = Rectangle(10,10,100,100)
r.draw()
r.move(5,5)
```


Redéfinition (1/2)

Dans l'exemple précédent, on redéfinit (*overwriting*) dans la classe `Rectangle` la méthode `draw` de la classe `Graphical`.

De même, on peut définir une classe `Circle` qui hérite de `Graphical` et redéfinit aussi la méthode `draw`

```
class Circle(Graphical):
    def __init__(self,x,y,r):
        self.r = r
        super().__init__(x,y,2*r,2*r)
    def draw(self):
        print("je dessine un cercle")
        return
```


Redéfinition (2/2)

L'intérêt de l'héritage et des redéfinitions s'illustre alors facilement.

```
def dessiner(t):
    for i in range(0,len(t)):
        t[i].draw()

t = [Rectangle(10,10,100,100), Circle(50,50,10)]
dessiner(t)
```

Classes abstraites (1/2)

On peut remarquer qu'il n'a pas lieu de créer des instances de la classe `Graphical` ; c'est ce qu'on appelle une classe abstraite.

En effet, certaines méthodes comme `draw` ne sont pas fournies et elles doivent être redéfinies dans les sous-classes.

On peut formaliser cela en utilisant les metaclass de Python.

```
from abc import ABCMeta, abstractmethod

class Graphical(metaclass=ABCMeta):

    def __init__(self,x=0,y=0,w=0,h=0):
        self.x = x; self.y = y;
        self.width = w; self.heigth = h

    def move(self,dx,dy): self.x += dx; self.y +=dy

    @abstractmethod
    def draw(self): pass
```


Classes abstraites (2/2)

Ainsi, il n'est pas possible de créer un objet en instantiant directement la classe `Graphical`.

Par exemple, la déclaration `p = Graphical()` va provoquer l'erreur suivante :

```
Traceback (most recent call last):
  File "cours4.py", line 13, in <module>
    p = Graphical()
TypeError:
  Can't instantiate abstract class Graphical with abstract methods draw
```

Il est alors obligatoire de redéfnir la méthode `draw` dans toutes les classes qui héritent de `Graphical`. Autrement, on obtient une erreur similaire pour les objets de ces nouvelles classes.

Typage

Les classes définissent de nouveaux types de données.

Par exemple, lorsqu'on déclare

```
class A:
    "classe A"

p = A()
print(type(p))
```

Le type de `p` est `<class '__main__.A'>`

On peut également savoir si un objet appartient à une classe :

```
isinstance(p,A)
```


Sous-Typage

La notion d'héritage s'accompagne d'une notion de sous-typage : un objet d'une classe `B` peut être vu comme un objet d'une classe `A`, si `B` hérite de `A`.

```
class A:
    def __init__(self,x):
        self.x = x

def f(p):
    assert(isinstance(p,A))
    return p.x + 1

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

p = B(10)
print (f(p))
```

Exos

Exos sur les piles

Voici quelques opérations qu'une structure de pile doit supporter :

```
create()        crée une pile vide
is_empty(s)     renvoie vrai si la pile s est vide
size(s)         renvoie le nombre d'éléments dans la pile s
push(x,s)       ajoute un élement x à la pile s
pop(s)          dépile et renvoie l'élément en haut de pile s
peek(s)         renvoie l'élément en haut de la pile s
...
```


Exercices 1

Implémenter une classe `Stack` pour réaliser cette structure de pile.

Exercices 2

Écrire une fonction `convert(n)` qui convertit un entier `n` (supposé positif) en binaire (sous forme d'une liste de `0` et `1`).

Exercices 3

Écrire une fonction `check(s)` qui permet de déterminer si une chaîne de caractères `s` constituée uniquement des caractères `(` et `)` est bien parenthésée.

Par exemple,

```
check('(())()(())')
 True
check('(())()(()')
 False
```


Exercices 4

Étendre la fonction précédente afin de vérifier une chaîne de caractères formées des caractères `(`, `)`, `[` et `]`.

Exercices 5

Écrire une fonction `compute` pour calculer le résultat d'une expression artithmétique en polonaise inverse (comme dans certaines calculatrices).

Dans cette notation, les opérateurs sont placés après leurs opérandes, en notation post-fixe. L'intérêt est que les parenthèses sont inutiles.

Pour simplifier, on supposera que l'expression ne contient que deux opérateurs `+` et `*` ainsi que des entiers entre 0 et 9.

Par exemple,

```
print (compute('12+3*4+'))
13
```


Exos sur les files

Une file est une structure où les éléments sont retirés dans l'ordre d'arrivée, ce qui correspond exactement à la notion usuelle de file d'attente.

On peut également associer une priorité aux éléments, qui ne sont alors plus retirés selon l'ordre d'arrivée ; on parle de file de priorité.

Opérations sur les files

Voici quelques opérations qu'une structure de file doit supporter :

```
create()        crée une file vide
is_empty(q)     renvoie vrai si la file s est vide
size(q)         renvoie le nombre d'éléments dans la file q
enqueue(x,q)    ajoute un élement x à la file q
dequeue(q)      dépile et renvoie l'élément en haut de file q
...
```


Exercice 6

Implémenter une classe `Queue` pour réaliser cette structure de file.