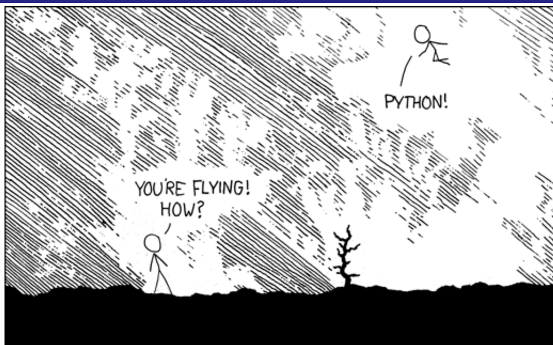
Le langage Python

La GUI tkinter

Christian Nguyen

Département d'informatique
Université de Toulon et du Var

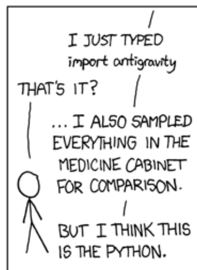
2012-2013



YOU'RE FLYING! HOW?
PYTHON!

I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"

I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?



I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

Présentation

Python est né en 1990, son concepteur est De Guido Van Rossum(Pays Bas).

Il présente les caractéristiques suivantes :

- syntaxe très simple,
- typage dynamique,
- portable,
- sous licence libre,
- extensible,
- multi-paradigme (impératif et orienté objet),
- interprété ou pré-compilé (*byte code*).

Utilisation

Deux modes d'utilisation :

- en mode interactif : dialogue avec l'interprète, chaque instruction est interprétée puis exécutée ("shell" de Python),
- créer des scripts (programmes) soit grâce à un éditeur de texte soit grâce à un environnement de développement intégré (IDE) par exemple IDLE. L'exécution du script se fait via l'interprète.

Mots réservés

33 mots :

- structures de contrôle : break, continue, if, elif, else, for, in, while, pass,
- expressions logiques : and, not, or, True, False,
- modularité : class, def, global, nonlocal, return, from, import,
- exceptions : try, except, finally, raise,
- autre : as, assert, del, is, lambda, None, with, yield

Données et variables

Les noms des identificateurs doivent respecter des règles de syntaxe (1er caractère lettre ou *underscore*, pas de caractères spéciaux, pas de mot réservé du langage, ...).

Syntaxe de l'affectation : `<identificateur de variable> = <expression>`

```
>>> x = 10
>>> mot = 'toto' # ou "toto"
```

Opérateurs

Les opérateurs symbolisent les opérations mathématiques :

- opérateurs arithmétiques : +, -, /, //, *, **, %
- opérateurs de comparaisons : <, >, <=, >=, ==, !=
- opérateurs logiques : and, or, not

Exemple :

```
>>> 2**3
8
>>> 5//2
2
>>> 5%2
1
>>> (2 < 3) and (3 != 7)
True
```

Expressions

Une expression est obtenue en combinant : valeurs, identificateurs et opérateurs, en respectant des règles de syntaxe précises.

L'interprète évalue une expression pour obtenir une *valeur unique* (ainsi qu'un type).

- expression arithmétique,
- expression booléenne.

```
>>> a+2

>>> 2 < 3

>>> 1 < 2 < 3

>>> ok and (a < 5)
```

Priorité des opérateurs

Expression qui contient plusieurs opérateurs : l'évaluation se fait de la gauche vers la droite si les opérateurs ont même priorité.

- opérateurs arithmétiques : PEMDAS (parenthèses, exposant, multiplication, division, addition et soustraction),
- opérateurs de comparaison : $<$, $<=$, $>$, $>=$, $!=$ et $==$,
- opérateurs logiques : or, and et not.

Priorités : les parenthèses puis les opérateurs arithmétiques, de comparaison et enfin logiques.

Typage des variables

- le type `int` : les entiers relatifs,
- le type `bool` : type booléen (deux valeurs : True et False),
- le type `float` : les nombres à virgule flottante (approximation des réels), ils se distinguent de `int` par la présence du point décimal (.), représentés par $s.m.b^e$ avec s signe, m mantisse, b base et e exposant (en faisant varier e on fait "flotter" la virgule décimale),
- le type `str` : les chaînes de caractères, spécifiées soit entre apostrophes, soit entre guillemets, exemple : `mot = 'bonjour'` ou `mot = "bonjour"`.

Conversion de type

Il existe plusieurs fonctions qui permettent de forcer le type d'une variable en un autre type.

- `int()` : permet de modifier une variable en entier, provoque une erreur si cela n'est pas possible,
- `float()` : permet la transformation en flottant,
- `str()` : permet de transformer la plupart des variables d'un autre type en chaînes de caractère.

Exemple

```
>>> int(3.14)
3
>>> str(3.14)
'3.14'
```

Les entrées / sorties

- fonction `input()` : retourne un objet de type str saisi au clavier.
- fonction `print()` : affiche l'expression passée en paramètres.

```
>>> a = input()
'2'
>>> print(a)
'2'
>>> a = int(input())
2
>>> a = float(input())
2.0
```

Structures de contrôle
Séquence et bloc

Les instructions s'exécutent les unes après les autres dans l'ordre d'écriture.

Dans les instructions composées, une séquence d'instructions s'appelle un bloc d'instructions.

L'indentation détermine les instructions d'un même bloc.

Il est recommandé de commenter chaque *bloc* d'instructions afin d'expliquer en langage clair son rôle et son fonctionnement.

Structures de contrôle
Sélection

Syntaxe

if <condition>:
    <bloc_d_instructions>
[elif <condition>:
    <bloc_d_instructions>]
[else:
    <bloc_d_instructions>]

```
a = 11
if a > 10:
    print("a est plus grand que dix")
elif a == 10:
    print("a est egal a dix")
else:
    print("a est plus petit que dix")
```

Structures de contrôle
Itération

Syntaxe

while <condition>:
    <bloc_d_instructions>

```
>>> a = 0
>>> while (a < 7):    # (n'oubliez pas le double point !)
...     a = a + 1     # (n'oubliez pas l'indentation !)
...     print(a, end=' ')
>>> 1 2 3 4 5 6 7
```

Structures de contrôle

Itération sur une séquence

Syntaxe

for <ident> **in** <iterateur>**:**

    <bloc_d_instructions>

```
>>> module = 'IHM'
>>> for i in module:
...     print(i, end='.')
>>> I.H.M.
```

```
>>> for i in range(7): # (ou (1, 2, 3) par exemple)
...     print(i)
>>> 0 1 2 3 4 5 6
```

Script Python

C'est un fichier texte. Son déroulement est linéaire sauf structure de contrôle (sélection ou itération).

Remarque : dans un script, les commentaires sont les lignes qui commencent par le caractère #.

Pour que l'exécution d'un script Python se fasse de façon implicite, il faut faire suivre le shebang `#!` du chemin d'accès à l'interprète (et rendre le fichier exécutable).

Définition d'une fonction
Fonction simple

Une fonction accepte de 0 à n paramètre(s) et retourne ou non un résultat.

Définition

def <nom_fonction>([<paramètres>]):
<bloc_d_instructions>

Instruction permettant de retourner un résultat : `return` (une fonction retourne par défaut `None`).

Instruction vide (prototypage) : `pass`

Définition d'une fonction

Fonction simple

```
>>> def f():
...     x=1
>>> y=f()
>>> print(y)
None
```

Une fonction peut retourner plusieurs résultats (un tuple).

```
>>> def f(px, py):
...     return px+1, py*10
>>> x, y = 1, 1
>>> x, y = f(x, y)
>>> print(x, y)
2 10
```

Visibilité, durée de vie
Variables globales vs locales

Visibilité :

- une variable globale est définie en dehors de toute fonction,
- une variable locale est définie dans une fonction et masque toute autre variable portant le même nom

Durée de vie :

- une variable globale existe durant l'exécution du programme,
- une variable locale existe durant l'exécution de la fonction.

Variables globales

Attention danger

Sans mécanisme de protection, la manipulation directe des variables globales est une hérésie à cause des *effets de bords*.

Dans un langage interprété, les variables sont locales à un bloc par défaut. Pour changer la portée : mot-clé `global`.

Exemple :

```
>>> x=10
>>> def f():
...     x=12
>>> f()
>>> print x
10
```

Passage de paramètres
Mutables ou non mutables

Quelques types sont immutables : les nombres, les chaînes de caractères, les tuples et les “frozensets”. Les autres types sont mutables.

Conséquences :

- le passage de paramètres effectifs de type immutable n’a pas d’incidence sur ceux-ci, la fonction concernée ne peut les modifier,
- le passage de paramètres effectifs de type mutable autorise la fonction à les modifier (!).

Passage de paramètres
Valeur par défaut

Possibilité de définir un argument par défaut pour chaque paramètre. *Attention* : les paramètres sans valeur par défaut doivent précéder les autres paramètres.
Conséquence : la fonction peut être appelée avec une partie de ses arguments.

Exemple :

```
>>> def Question(mesg, rep=('o','O','n','N')):
...     choix = ''
...     while not choix in rep:
...             print "La réponse doit être dans "+str(rep)
...             choix = raw_input(mesg)
...     return choix
```

Passage de paramètres
Arguments avec étiquettes

Les paramètres peuvent être transmis dans un ordre quelconque si tous les paramètres ont une valeur par défaut dans la définition de la fonction et à condition de désigner nommément les paramètres.

```
>>> def Cercle(x=0.0, y=0.0, r=1.0):
...
>>> Cercle(r=7.7, x=1.3, y=5.0)
```

Fichier

Séparation des données et du programme dans des fichiers séparés.

Un fichier est une ressource critique : les opérations sur les fichiers passent par le système.

3 étapes fondamentales :

1. ouvrir le fichier,
2. lire ou écrire dans ce fichier,
3. fermer ce fichier.

Fichier
Ouverture

L'ouverture d'un fichier peut entraîner sa réservation à l'usage exclusif du demandeur.

Définition

<desc> = open(<nom_du_fichier>, <mode_d_ouverture>)

Il retourne un objet particulier : le descripteur du fichier.

Les modes d'ouverture sont : "r" (read), "w" (write) ou "a" (append).

Fichier
Lecture/écriture

Les deux opérations se font de façon *séquentielle*.

La lecture d'un fichier est a priori non bloquante. Elle s'arrête à la fin du fichier (`read()` renvoie une chaîne vide).

Définition

<desc>`.read(`[<nombre_de_caractères>]`)`

A contrario, l'écriture dans un fichier est obligatoirement bloquante. Les données à écrire doivent être fournies en argument.

Définition

<desc>`.write(`<données>`)`

Fichier
Fermeture

Cette opération correspond essentiellement à la libération de la ressource.

Mais il a aussi comme fonction de garantir le traitement des dernières données (utilisation d'une mémoire tampon).

Définition

<desc>`.close()`

Fichier
Fichier texte

Contient des caractères imprimables et des espaces organisés en lignes successives, séparées par des retours à la ligne (caractère spécial `\n`).

Création, par exemple :

```
f.write("Ceci est une ligne\n")
```

Lecture, par exemple :

```
ligne = f.readline()
```

Types composites
Tuple

C'est une liste (séquence, suite ordonnée) *non modifiable*, une collection d'éléments séparés par des virgules. Il est conseillé de l'encadrer par une paire de parenthèses ( ).

Exemple :

```
tuple = ('a', 'b', 'c')
```

Accès : par un index entre crochets, par exemple `tuple[0]`.

Avantages : moins de ressources, plus efficace.

Types composites
Chaîne de caractères

Ce sont des tuples de type *string* définis par une suite quelconque de caractères, ces derniers délimités par des apostrophes ou des guillemets. Exemple :

```
chaine = 'Bonjour le monde'
```

Opérations élémentaires :

- assembler plusieurs chaînes à l'aide de l'opérateur `+`,
- déterminer la longueur d'une chaîne en faisant appel à la fonction `len()`,
- convertir une chaîne en nombre (`int()` ou `float()`) ou vice-versa (`str()`).

Types composites
Liste

C'est une liste (séquence, suite ordonnée) *modifiable* d'éléments séparés par des virgules, encadrés par une paire de crochets [ ].

```
liste = [1, 'a', 3.14]
```

Accès : par un index entre crochets, par exemple `liste[0]`.

Opérations élémentaires :

- ajouter un objet à la fin : `L.append(obj)`,
- nombre d'occurence d'un objet : `L.count(obj)`,
- premier indice d'un objet : `L.index(obj)`,
- insertion d'un objet avant l'index *i* : `L.insert(i, obj)`,
- tri (sur place) d'une liste : `L.sort()`.

Liste
Technique de *slicing*

Cette syntaxe permet de manipuler les listes de façon intuitive et efficace.

Syntaxe

<liste>[<debut>:<fin>]

Exemples :

```
>>> ll = [1, 2, 3, 4, 5]
>>> ll[:2]
[1, 2]
>>> ll[4:]
[5]
>>> ch = 'IHM'
>>> ch[1:2]
'H'
```

Liste

Les ‘*list comprehensions*

Listes dont le contenu est défini par filtrage du contenu d’une autre liste.

Exemples :

```
>>> [x**2 for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> [m[0] for m in ('un', 'deux', 'trois')]
['u', 'd', 't']
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Types composites
Ensemble

Un ensemble est une collection non ordonnée d'éléments uniques.
Les opérateurs ensemblistes union, intersection, différence et différence symétrique sont associés à ce type.
Exemples :

```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a, b
({'a', 'r', 'b', 'c', 'd'}, {'a', 'c', 'z', 'm', 'l'})
>>> a | b                       # union
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                       # intersection
{'a', 'c'}
>>> a - b                       # difference
{'r', 'd', 'b'}
>>> a ^ b                       # difference symetrique
{'r', 'd', 'b', 'm', 'z', 'l'}
```

Types composites
Dictionnaire (ou tableau associatif)

Ce ne sont pas des séquences, cependant les éléments sont accessibles par une *clé* (numérique, alphabétique, composite).

Syntaxiquement, un dictionnaire est constitué d'éléments enfermés dans une paire d'accolades. Un dictionnaire vide sera donc noté {}.

Exemple :

```
couleurs = {}
couleurs['jaune'] = 'yellow'
couleurs['vert'] = 'green'
# ou bien
couleurs = {'jaune' : 'yellow', 'vert' : 'green'}
```

Types composites
Dictionnaire

Opérations élémentaires :

- copier un dictionnaire : `D.copy()`,
- avoir les clés sous forme de liste : `list(D.keys())`,
- avoir les valeurs sous forme de liste : `list(D.values())`,
- savoir si un élément est dans le dictionnaire : opérateur `in`,
- supprimer un élément : `del()`

Programmation modulaire
Généralités

Toutes les fonctions ne sont pas intégrées au langage.

Besoins :

- développement réparti (entre plusieurs équipes),
- réutilisabilité (utiliser en adaptant dans un contexte similaire).

Solutions :

- encapsulation (organisation du code en unités logiques),
- protection des données (visibilité),
- espaces de noms (préfixe, levée des ambiguïtés).

Modules

Mise en œuvre avec Python

Définition de données et de fonctions dans un même fichier.

Importation du module par le nom du fichier, exemple :

```
import math   # utilisation : math.sqrt(n)
from math import *   # utilisation : sqrt(n).
```

Définition de données ou de fonctions "privées" : nom préfixé de deux blancs soulignés (__), par exemple :

```
##### fonction privee d'affichage d'une erreur
def __erreur(perr):
    print(perr, file=sys.stderr)
```

Exceptions

Capture et traitement des exceptions : le couple `try`, `except`

```
>>> try:
...     x=3/0
... except:
...     print('division par zero')
...
division par zero
```