Introduction à Python et prise en main de l'environnement

DIU Enseigner l'Informatique au Lycée

G. Huard, L. Mounier, C. Parent-Vigouroux, A. Rasse, B. Wack

UFR IM2AG, Université Grenoble Alpes

avril 2019

Remerciements : Stéphane Gonnord, Matthieu Moy, Marc de Falco

Plan

Plan

Python : en quelques points

- Un langage de script
- **Langage interprété** ($\neq$ langage compilé)
- **Typage Dynamique** ($\neq$ typage statique)
- **Indentation significative**
- **Orienté objet**
- Gestion automatique de la mémoire (*garbage collector*)

Approche *"batteries included"* :

- Bibliothèque standard Python : plus de 200 packages allant de simples structures de données à des primitives d'accès web ou de traitement multimedia
  `http://docs.python.org/3/library/`
- De fait, un langage vite adopté par les entreprises et intégré dans les systèmes d'exploitation

Points forts, points faibles

- Dans la vraie vie :
  - Langage de haut niveau : on peut faire beaucoup avec peu de code
  - Typage dynamique $\Rightarrow$ lent et gourmand en mémoire
  - Écosystème très fourni
  - Facile à apprendre, mais intéressant aussi pour des experts

Points forts, points faibles

- Dans la vraie vie :
  - Langage de haut niveau : on peut faire beaucoup avec peu de code
  - Typage dynamique $\Rightarrow$ lent et gourmand en mémoire
  - Écosystème très fourni
  - Facile à apprendre, mais intéressant aussi pour des experts
- Pour la pédagogie :
  - Démarrage en douceur, mais il y a un cap à franchir par la suite
  - Typage dynamique : discutable
  - Indentation obligatoire : vos élèves sont obligés d'écrire du code lisible (ou presque)

Dynamique de programmation avec Python

- Python dispose d'un interpréteur interactif :

```
>>> 2 + 2
4
```

expression Python, entrée par l'utilisateur

réponse de l'interpréteur après évaluation

Dynamique de programmation avec Python

- Python dispose d'un interpréteur interactif :

```
>>> 2 + 2      expression Python, entrée par l'utilisateur
    4          réponse de l'interpréteur après évaluation
```

- Utilisable en tant que "calculatrice" mais vite limité, notamment si on veut écrire un programme ou modifier son travail

Dynamique de programmation avec Python

- Python dispose d'un interpréteur interactif :

```
>>> 2 + 2      expression Python, entrée par l'utilisateur
    4          réponse de l'interpréteur après évaluation
```

- Utilisable en tant que "calculatrice" mais vite limité, notamment si on veut écrire un programme ou modifier son travail
- Concrètement, on écrira plutôt les programmes dans un éditeur de texte, le plus souvent inclus dans un Environnement de Développement Intégré (Spyder, IDLE, Pyzo...)

Dynamique de programmation avec un IDE

1. On écrit son programme, de préférence sous forme de **fonctions**
   - pas d'`input` ni de `print`
   - mais des arguments et une valeur de retour

Dynamique de programmation avec un IDE

1. On écrit son programme, de préférence sous forme de **fonctions**
   - pas d'`input` ni de `print`
   - mais des arguments et une valeur de retour
2. On fait évaluer les fonctions par l'interpréteur (mais elles ne sont pas encore *exécutées*)

Dynamique de programmation avec un IDE

1. On écrit son programme, de préférence sous forme de **fonctions**
   - pas d'`input` ni de `print`
   - mais des arguments et une valeur de retour
2. On fait évaluer les fonctions par l'interpréteur (mais elles ne sont pas encore *exécutées*)
3. On teste ses fonctions avec des paramètres réels dans l'interpréteur, qui permet d'observer les valeurs renvoyées

Dynamique de programmation avec un IDE

1. On écrit son programme, de préférence sous forme de **fonctions**
   - pas d'`input` ni de `print`
   - mais des arguments et une valeur de retour
2. On fait évaluer les fonctions par l'interpréteur (mais elles ne sont pas encore *exécutées*)
3. On teste ses fonctions avec des paramètres réels dans l'interpréteur, qui permet d'observer les valeurs renvoyées

3bis. On peut aussi placer des tests d'appels aux fonctions dans le code source mais attention, pour que le résultat s'affiche il faut un `print`

Plan

Les variables

- Les variables ne se déclarent pas, elles sont définies à partir de leur première affectation

```
>>> x = 42
>>> x + 1
43
>>> x = y + 1
NameError: name 'y' is not defined
```

Les variables

- Les variables ne se déclarent pas, elles sont définies à partir de leur première affectation

```
>>> x = 42
>>> x + 1
43
>>> x = y + 1
NameError: name 'y' is not defined
```

- Typage dynamique

```
>>> x + 1
43
>>> x = [1,2,3]
>>> x + 1
TypeError: can only concatenate list
    (not "int") to list
```

Types de données de base

- Entiers *longs* illimités : `0`, `-4`, `42`, `123456789000987654321`
  - opérations usuelles dont `//` (quotient euclidien) et `**` (exponentiation)
- Flottants : `0.0`, `0.5`, `.5`, `-1.`, `1.2e+20`
  - pas « `0,2` » !
  - opérations usuelles dont `/` (quotient décimal)
  - précision liée à la machine, en général 64 bits
- Conversions implicites automatiques lorsque c'est nécessaire (par exmeple pour évaluer `42/17`)

Types de données de base (suite)

- Chaînes de caractères : trois formes
  - `"Bonjour"`
  - `'Au revoir'` (équivalentes)
  - `"""Rebonjour"""` (autorise les retours à la ligne)
- Booléens : `True` et `False`
  - opérateurs logiques : `and or not`
  - relations de comparaisons usuelles `== != < > <= >=`
- Pour convertir explicitement une valeur dans un autre type :

```
>>> int(3.14)
3
>>> str(42)
'42'
```

Utilisation de librairies

Exemple : fonctions mathématiques dans le module `math`

- Deux possibilités d'accès :
  - Import du module et accès depuis le module

```
>>> import math
>>> math.sqrt(2.0)
1.4142135623730951
```

  - Import spécifique de la fonction et accès direct

```
>>> from math import sqrt
>>> sqrt(2.0)
1.4142135623730951
```

- Listing des fonctions et constantes : `dir(math)`
- Aide embarquée : `help(math.ceil)`
  ou même `help(math)`

Plan

Fonctions

```
def moyenne(a, b):    # définition de la fonction
    s = a + b
    return s / 2

m = moyenne(42, 3)    # appel à la fonction
m = moyenne(x, 1)
```

- corps de la fonction indenté
- pas de type pour les arguments ni pour la valeur de retour
- par défaut : `return None`
- Attention à la délimitation des blocs : un `return` mal indenté peut coûter très cher...

Condition : if/then/else

- Si alors

```
def abs(x):
    if x < 0:      # ne pas oublier le ':'
        x = -x
    return x       # indentation signifiante
```

- Si alors sinon

```
def abs(x):
    if x < 0:
        return -x
    else:
        return x
```

Boucles

- Boucle conditionnelle (classique)

```
while x <= 42:
    x = x + 1       # indentation signifiante
```

Boucles

- Boucle conditionnelle (classique)

```
while x <= 42:
    x = x + 1       # indentation signifiante
```

- Boucle inconditionnelle (particulière en Python, mais à l'usage plus pratique à utiliser)

```
for variable in iterable:
    instructions
```

Boucles

- Boucle conditionnelle (classique)

```
while x <= 42:
    x = x + 1       # indentation signifiante
```

- Boucle inconditionnelle (particulière en Python, mais à l'usage plus pratique à utiliser)

```
for variable in iterable:
    instructions
```

- Qu'est-ce qu'un itérable ?
  Liste, chaîne... :

```
s = 0
for e in [1,2,3]:
    s += e
for c in 'abc':
    print(c)
```

Un itérable pratique

- Parcourir les nombres de 0 a $N-1$:

```
for i in range(10):
    print(i)
# Affiche les nombres de 0 a 9 inclus
```

- Parcourir les nombres de M à $N-1$:

```
for i in range(7, 10):
    print(i)
# 7
# 8
# 9
```

- Avec un pas :

```
for i in range(0, 10,2):
    print(i)
# Affiche 0, 2, 4, 6, 8
```

Plan

Quelques astuces à connaître

- Commentaires :
  Tout ce qui suit un # sur une ligne est ignoré
- Affectation multiple simultanée

```
x, y = 14, 42
```

Quelques astuces à connaître

- Commentaires :
  Tout ce qui suit un # sur une ligne est ignoré
- Affectation multiple simultanée

```
x, y = 14, 42
```

```
x, y = y, x
```

Les entrées / sorties

- La saisie clavier :

```
# Lire une chaine
ch = input("Entrez une chaine : ")

# Lire un entier
# (lecture de chaine puis conversion)
nb = int(input("Entrez un nombre : "))
```

- Affichage :

```
print("toto")
print(42)
print("toto", 42)
```

- Il y a une conversion automatique des arguments en `str`.

Plan

Les listes Python

- éléments entre `[ ]` séparés par des `,`

- Hétérogènes
- Dimension donnée par `len`
- Accès aléatoire, **possible en arrière**
- **Modifiables** (on dit aussi *mutable*)

```
>>> L = [1,True,3]
>>> len(L)
3
>>> L[0]
1
>>> L[-1]
3
>>> L[-2]
True
>>> L[0] = 42
>>> L
[42, True, 3]
```

Plan

Mais aussi...

Ajout en queue append

- Prend l'élément à rajouter en argument
- Modifie le tableau
- Ne renvoie rien

```
>>> L = [42, True, 3]
>>> L.append(4)
>>> L
[42, True, 3, 4]
```

Retrait en queue pop

- Ne prend pas d'argument
- Modifie le tableau
- Renvoie l'élément retiré

```
>>> L.pop()
4
>>> L
[42, True, 3]
```

Mais aussi...

Ajout en queue append

- Prend l'élément à rajouter en argument
- Modifie le tableau
- Ne renvoie rien

```
>>> L = [42, True, 3]
>>> L.append(4)
>>> L
[42, True, 3, 4]
```

Retrait en queue pop

- Ne prend pas d'argument
- Modifie le tableau
- Renvoie l'élément retiré

```
>>> L.pop()
4
>>> L
[42, True, 3]
```

En Python, on parle donc plutôt de listes que de tableaux.

Quelques explications

Une liste est un objet

- La plupart des opérations de manipulation sont donc des méthodes.

```
>>> nomdelaliste.nomdelamethode(...)
```

- Comme pour les modules :

```
>>> dir(maliste)                # liste des methodes
>>> help(maliste.methode) # doc d'une methode
```

Attention : ce ne sont pas des listes chaînées

- accès et modification aléatoire en temps constant
- ajout et suppression en temps constant amorti

Plan

Exemple : extraction d'une sous-liste

Rappel : quand on écrit `for x in a`, la variable x prend successivement toutes les valeurs des éléments de a.

Seuil

```
def seuil(L, s):
    M = []
    for x in L:
        if x > s:
            M.append(x)
    return M
```

Plan

Attention au passage de paramètres

En Python, *tout* est référence :

```
def f(t):
    t[2] = 0

a = [1,2,3,4,5]
f(a)
print(a)      # [1,2,0,4,5]
```

pour tous les types de données mutables.

Comparaison mutable / non-mutable

Version int

```
>>> def f(x):
        x = x + 1


>>> a = 3
>>> f(a)
>>> a
3
```

Version liste

```
>>> def g(L):
        L.append(0)


>>> M = [1, 2, 3]
>>> g(M)
>>> M
[1, 2, 3, 0]
```

Comparaison mutable / non-mutable

Version int

```
        # id(x)=id(a)
>>> def f(x):
        x = x + 1


>>> a = 3
>>> f(a)
>>> a
3
```

Version liste

```
        # id(L)=id(M)
>>> def g(L):
        L.append(0)


>>> M = [1, 2, 3]
>>> g(M)
>>> M
[1, 2, 3, 0]
```

Comparaison mutable / non-mutable

Version int

```
>>> def f(x):
        x = x + 1
      # modifie id(x)
>>> a = 3
>>> f(a)
>>> a
3
```

Version liste

```
>>> def g(L):
        L.append(0)
      # ne modifie pas id(L)
>>> M = [1, 2, 3]
>>> g(M)
>>> M
[1, 2, 3, 0]
```