Programmation 101

Pier-André Bouchard St-Amant

École nationale d'administration publique

2 mars 2018

De quoi s'agit-il ?

- Développer une compréhension de la construction d'algorithmes par blocs.
- Langage de programmation utilisé : Python.
- Avantages de Python : *open-source* (gratuit), répandu et donc bien outillé.
- Ceci n'est pas un toturiel avancé de programmation, ni un cours exhaustif sur Python.

Qu'est-ce qu'un programme/algorithme ?

- Les ordinateurs comprennent un seul langage : les zéros et les un.
- Un programme est une séquence finie de 0 et de 1 (instructions), qui sont lues par l'ordinateur afin de produire une sortie (une autre séquence de 0 et de 1).
- Un programme sert à écrire algorithme, soit des instructions qui, étant données la même entrée, vont donner la même sortie.
- Le programme est écrit sur la machine (concret), alors que l'algorithme est le "schéma" du programme (abstrait).
- Un algorithme peut donc être écrit en français (ou pseudo-code), tant qu'il est assez précis pour qu'une même entrée donne toujours une même sortie.

Qu'est-ce qu'un langage de programmation ?

- Écrire des séquences de 0 et de 1 est peu pratique. Furent donc inventées des règles/mots-clés abstraits qui représentent une séquence de 0 et de 1. Ces règles/mot-clés forment les langages de programmation.
- Un programme peut donc être écrit à partir de ces mots-clés.
- Un compilateur/interpréteur convertit ces fichiers de texte en séquences de 0 et de 1, pour que l'ordinateur comprenne.
- Exemples de langages de programmation : c++, java, lisp, prolog, fortran, basic, python, Matlab, PHP, etc.

Étapes de la rédaction d'un programme

- Comprendre le problème à résoudre.
- Concevoir l'algorithme pour le résoudre (étape clé).
- Séparer la tâche en étapes.
- Rédiger le code (programme).
- Tester le code.

Quels sont les défis de la programmation pour les débutants ?

Trois principaux :

- Comprendre les composantes de base des algorithmes.
- Comprendre les mots-clés/conventions d'un langage de programmation.
- Concevoir les algorithmes.

Instructions de base derrière la programmation

- Les ordinateurs peuvent faire deux choses avec des données : les sauvegarder dans la mémoire ou effectuer des opérations sur ces dernières.
- Les données sont sauvegardées dans des variables.
- L'interpréteur lit les programmes de droite à gauche (sauf pour la priorité des opérations dans les parenthèses).

Affectation

- $x = 10$ est un exemple d'affectation d'une donnée (la valeur 10) à une variable (x).
- Ainsi, quelque part dans la mémoire vive de l'ordinateur, un espace spécifique est réservé pour cette variable, et la valeur est sauvegardée dans dans cet espace.
- Puisqu'on lit de droite à gauche, $x = x + 10$ est une expression qui a du sens : on prend la valeur sauvegardée dans x, on lui ajoute 10, et on sauvegarde le résultat dans x (effaçant la valeur précédente).

Opérations

- Les opérateurs de base existent : $+, -, \backslash, *$.
- Il existe des tonnes d'opérateurs. Ce qu'ils font exactement dépend du langage de programmation utilisé.
- Les opérateurs peuvent agir sur des nombres, chaîne de charactères, matrices, fichiers, etc.
- Ainsi, certaines opérations sont permises uniquement sur certains types de variables.
- Selon la précision du langage, un mauvais usage des opérateurs peut mener à un échec de la compilation/interprétation du code.
- Pire encore, il pourrait compiler, mais calculer des résultats qui n'ont pas de sens.

Variables en Python

- En Python, toute chaîne qui commence avec une lettre peut être utilisée comme variable : *toto*, x, *a_b*, *tomate*21, etc.
- Les variables auxquelles sont assignées des valeurs se voient automatique déclarer un **type** de variable :
  - *int* : entier
  - *float* : nombre décimal
  - etc.

  Ainsi, si on écrit $x = 1$, x sera automatiquement de type entier, alors que si on écrit $x = 1.0$ ou $x = 1.1$, x sera de type décimal.

Opérations en Python

- Opérateurs de base : $+, -, /, *$
- Opérateurs de base non intuitifs :
  - $**$ sert d'exposant (ex. $2 * *2$ pour 2^2).
  - $x+ = 1$ ajoute 1 à x (équivalent à $x = x + 1$)
- Les opérateurs respectent le type de variable : $int/int{=}int$.
  - Donc ATTENTION : en python, $1/2 = 0$, mais $1.0/2.0 = 0.5$.
  - Il est possible de modifier le type d'une variable en utilisant la fonction appropriée :
    - d'entier à décimal : $x2 = float(x1)$
    - de décimal à entier : $x2 = int(x1)$

Tableaux

- $x = [0, 1, 2]$ affecte un tableau de valeurs à x.
- 2 façons principales de générer un tableau de données en python :
  1) Un tableau de tableaux
     - $x = [[1, 2], [3, 4]]$ génère le tableau

1	2
3	4

     à la variable x.
     - Pour aller chercher un élément d'un tableau : $x[i][j]$, où i est le numéro de la colonne et j est le numéro de la rangée. Ainsi, dans notre exemple ci-haut, $x[0][1] = 2$.
       L'indice des données commence à $\mathbf{0}$.
  2) Matrice
     - À l'aide de fonctions. Nous y reviendrons plus tard.

Forme de base d'un code

- Pas plus d'une affectation/instruction par ligne.
- Le fichier est lu du haut au bas.
- Le symbole '#' permet de mettre des commentaires : tout ce qui vient après ce symbole n'est pas compilé. Ceci permet d'écrire des notes sur le fonctionnement de notre code.

Instructions par bloc

- Parfois, on veut faire des opérations seulement si certaines conditions sont remplies.
- Pour ça, on aura besoin (1) d'un outil pour vérifier si des propositions sont vraies ou fausses : l'algèbre booléenne, et (2) d'une structure pour vérifier ces propositions.

Algèbre booléen

- Algèbre booléenne : seules valeurs possibles pour une variable : vrai ou faux.
- Comme toute algèbre, il y a des variables et des opérateurs.
- Une proposition, par exemple '$A > 0$', peut prendre deux valeurs : vrai ou faux.
- Nous présentons ici les opérateurs booléens.

Opérateur de négation (not ou !)

L'opérateur de négation change la valeur d'une proposition.

Table – Opérateur de négation

prop. 1	not prop. 1
True	False
False	True

Par exemple, not($x \geq 0$) est Vrai si $x < 0$.

Opérateur "et" (*and*)

L'opérateur "et" donne vrai seulement si les deux propositions sont vraies.

Table – Operator "et"

prop. 1	prop. 2	prop. 1 *and* prop. 2
True	True	True
True	False	False
False	True	False
False	False	False

Par exemple, $(x > 10)$ and $(x < 2)$ est Vrai si x n'est pas dans l'intervalle $[2, 10]$.

Operator "ou" (*or*)

L'opérateur "ou" donne faux seulement si les deux propositions sont fausses.

Table – Operator "or"

prop. 1	prop. 2	prop. 1 *or* prop. 2
True	True	True
True	False	True
False	True	True
False	False	False

Par exemple, $(x > 10)$ or $(x < 2)$ est Faux si x est dans l'intervalle $[2, 10]$..

Propositions de comparaison numérique

- Les propositions de base sont composées de comparaisons numériques
  - $>$, $>=$ (plus grand ou égal), $<$, $<=$ (plus petit ou égal), $==$ (est égal à, car il doit être différent de l'opérateur d'affectation "$=$"), not $=$ (n'est pas égal à).
- Par exemple, si $a = 10$ and $b = 1$, alors $a > b$ est vrai, $a >= b$ est vrai, $a < b$ est faux, $a <= b$ est faux, $a == b$ est faux et $a! = b$ est vrai.

Propositions composites

Avec ceci, on peut construire des propositions composites. Par exemple, (not p) or (q and p). Les parenthèses s'appliquent comme d'habitude.

Table – Exemple d'opérateurs composites

prop. 1	prop. 2	(2 and 1)	not 1	(not 1) or (2 and 1)
False	False	False	True	True
False	True	False	True	True
True	False	False	False	False
True	True	True	False	True

Instructions et structure

- La structure pour faire des opérations si certaines conditions sont remplies : par bloc.
- Pour changer le flux d'instruction, on utilise les mots-clés de contrôle : *if* , *while*, *for*.
- Dans ce cas, c'est l'**indentation** qui détermine la structure du code : chaque bloc est "indenté" (décalé d'au moins un caractère vers la droite) par rapport au bloc dans lequel il se situe.

Opérateur "if" (si)

Pseudo-code de base :

1. Début du code.
2. Si une proposition est vraie...
   2.1 exécuter cette partie du code.
3. Si plutôt cette autre proposition est vraie...
   3.1 exécuter cette autre partie du code.
4. Sinon...
   4.1 exécuter cette autre partie du code.
5. Reste du code.

Exemple

```
x = 10;
if( x > 7) :
        print('x est plus grand que 7')
elif(x == 7) :
        print('x est égal à 7')
else :
        print('Zut alors!')

x += 1
```

Ce programme va afficher 'x est plus grand que 7'.
"Elif" est un diminutif pour "else if". Remarquez l'indentation.

Opérator "for"

Pseudo-code de base :

1. Début du code.
2. Pour une variable dans un intervalle...
   2.1 exécuter cette partie du code.
3. Reste du code.

Exemple

```
x = 0
for i in range(0,10,1) :
        x += i

print x
```

Ce programme va écrire "55".
La fonction "range" crée un vecteur de 0 à 10, par bonds de 1. On peut aussi écrire simplement "range(10)".

Opérateur "while"

Pseudo-code de base :

1. Début du code.
2. Tant qu'une proposition est vraie...
   2.1 exécute cette partie du code.
3. Reste du code.

Exemple

```
x = 0
stop = 10
while (x <= stop)  :
    x += 1

print x
```

Le programme va afficher "11".

Fonctions

- Il est facile de faire des erreurs en programmation.
- Ainsi, si A, B, C sont des segments de code, $(A + B)C$ est plus sécuritaire à écrire que $AC + BC$.
- Puisque C est écrit seulement une fois, il y a un seul endroit où une erreur est possible.
- De plus, certains segments de code peuvent être des programmes en eux-mêmes.
- Les **fonctions** permettent de définir du code qui peut être appelé par un autre segment du code.

Fonctions

Appel à une fonction :

```
[out1, out2, ...] = func_name(arg1, arg2, ...)
```

Où "outX" sont les sorties et "argX" sont les arguments que prend la fonction.

Fonctions disponibles

Il existe de nombreuses fonctions intégrées au langage de programmation.

```
x = min(x) # Trouve le minimum du vecteur x
x = range(0,10,1) ; range(10) # Fait un vecteur
                    # de 0 à 10, par pas de 1
long = len(x); # Longueur de x
help(fonction) # Fichier d'aide
```

Pour la liste complète :
https ://docs.python.org/3.3/library/functions.html

Librairies de fonctions

De nombreuses librairies offrent un nombre étendu de fonctions.

- Il faut importer ces librairies
- Au début du programme, on écrit :

```
import nom_librairie
```

- Ensuite, on appelle la fonction en écrivant :

```
x = nom_librarie.fonction(y)
```

- Si le nom de la librairie est trop long, on peut lui attitrer un diminitif. Par exemple :

```
import numpy as np
x = np.sum(y)
```

- Ou, si on veut tout importer directement (mais on se bloque des noms de variables avec le nom des fonctions) :

```
from numpy import *
x = sum(y)
```

Librairies utiles

Liste de librairies utiles en Python :

- math : fonctions mathématiques
- numpy : plusieurs fonctions
  - np.zeros((10,10)) matrice 10x10 remplie de 0
- matplotlib.pyplot : graphiques
- pandas : plusieurs fonctions, dont import et création de fichiers Excel

Un exemple : retour aux matrices

1) On avait les tableaux :

```
tableau[i][j][k]
```

2) On peut aussi gérer des matrices de données facilement avec la librairie numpy.

```
import numpy as np
M = np.zeros((3,4,4)) # Crée une matrice de 0
                      # de taille (3,4,4)
```

Dans ce cas, on accède aux éléments de la matrice de la façon suivante, pour l'élément (i,j,k) :

```
element1 = M[i,j,k]
```

Un exemple : retour aux matrices

2) On peut alors assigner les valeurs qu'on veut à cette matrice avec des boucles :

```
for i in range(M.shape[0]) :
  for j in range(M.shape[1]) :
      for k in range(M.shape[2]) :

          M[i,j,k] = ... # assigner une valeur
```

Créer sa propre fonction

- On peut créer ses propres fonctions, afin d'alléger son code.
- Afin d'être accessible dans plusieurs programmes, elles doivent être dans des fichiers indépendants.
- Elles doivent être dans le même dossier que le fichier qui appelle la fonction.
- On importe la fonction comme on importe les librairies, en écrivant

```
import nom_ficher
```

Déclaration de fonctions

Dans le fichier "mafonction.py" :

```
def test(arg1, arg2) :

# du code

return out1, out2 # sorties
```

Ainsi, on l'appelera en faisant :

```
import mafonction as mf
sortie1, sortie2 = mf.test(x, y)
```

Un exemple simple

Soit une fonction :

```
def sommeDiagonale(x) :
  # x est une matrice carrée

  n = x.shape[0] # taille

  # Fait la sommes des éléments sur la diagonale
  for i in range(n) :
      somme = somme + x[n,n]

return somme
```

Un exemple simple

Alors, on peut faire :

```
M = np.zeros((6,6))      # déclare la matrice
for i in range(6) :
    for j in range(6) :
        M[i,j] = i*j     # assigne des valeurs

S = sommeDiagonale(M)
```

Trucs de programmation

- Penser avant de taper.
- Diviser pour reigner.
- Clarté avant vitesse.
- Une ligne de code = une ligne de commentaire.
- Code sans bug $\not\Rightarrow$ il fait ce que tu veux.
- En cas de *bug*, Google est ton ami.

Travailler en groupe

- Il y a deux choses pires que de lire ton code : lire ton code un mois plus tard et lire le code de quelqu'un d'autre.
- Ainsi, travailler en groupe est beaucoup mieux quand on travaille avec des algorithmes abstraits, des spécifications et des tests.

Travailler en groupe

- Les algorithmes et les pseudo-codes permettent de connaître les idées générales à implémenter sans regarder le code.
- Les fonctions permettent de cacher du code. Une fois que les arguments, sorties et les conditions générales sont spécifiées dans un groupe, chacun peut coder seul.
- Ainsi, quelqu'un peut programmer le fichier principal, pendant que quelqu'un d'autre code des fonctions qui seront utilisées par ce dernier.
- Tester permet de déceler des erreurs dans les fonctions.

Un exemple

Un exemple : tracer et analyser des données à partir d'un fichier Excel. On a les données de la consommation de carotte par groupe d'âge et à diverses années dans le fichier exemple1.xlsx.

1. Importer les données dans une matrice.
2. Déclarer le vecteur qui contiendra les moyennes par année.
3. "For" chaque année :
   3.1 Faire la moyenne sur les groupes d'âge.
4. Tracer les données.

Un exemple

```
# Importer les librairies nécessaires
import pandas as pd                  # fichiers Excel
import matplotlib.pyplot as plt # graphiques
import numpy as np

# Importer le ficher excel
carottes = pd.read_excel('exemple1.xlsx')

# Mettre les données dans une matrice :
M = carottes.as_matrix()
# Extraire les en-tête en x dans un vecteur :
annees = carottes.columns

...
```

Le fichier Excel doit être dans le même dossier que le programme.

Un exemple

```
...
# Faire la moyenne par année
nb_annee = M.shape[1] # Nombre de colonnes

moyenne = np.zeros(nb_annee) # déclarer
for j in range(nb_annee) : # Pour chaque année
    moyenne[j] = np.mean( M[:,j] )

# Tracer les résultats
plt.plot(annees, moyenne)
plt.title(u'Carottes par semaine')
plt.xlabel(u'Année') ; plt.ylabel(u'g')
plt.show()
```

Le 'u' encode le texte en unicode, qui permet les caractères français.
Le ' ;' permet de mettre deux commandes simples par ligne.

Un exemple

On peut aller un peu plus loin et créer une matrice tri-dimensionnelle, avec des données sur la consommation de pommes.
On a donc 3-D : les années, les groupes d'âge, et le produit consommé.

Un exemple

```
# Importer les données
pommes = pd.read_excel('exemple1.xlsx',
    sheetname='Pommes')

# Mettre les données dans une matrice :
P = pommes.as_matrix()

# Créer la matrice 3D
Aliments = np.stack((M,P))
# Aliments(type d'aliment, groupe d'âge, année)

print 'La matrice Aliment est de forme',
    Aliments.shape
```

Travailler en groupe

Il faut aussi bien expliquer le comportement de ses fonctions.

- Comment est organisé le fichier qui contient les données ? Que se passe-t-il si l'importation des données échoue ?
- Que se passe-t-il si l'argument d'une fonction est vide ? S'il n'est pas de la bonne taille ?

Généralement :

- Quels sont les formats des arguments et les sorties dans ces cas ?
- Dans quelles circonstances la fonction fonctionne ?

Installation

Pour installer Python : https ://www.python.org/downloads/

Outils de programmation

Pour programmer, il nous faut...

- Un fichier texte ou écrire le programme.
  On sauvegarde le programme avec l'extension ".py"
- Un terminal pour rouler le programme

Pour accéder au fichier où se trouve le code, dans le terminal :

```
cd Directory1/Directory2/
```

Pour exécuter le code, on tape dans le terminal :

```
python fichier.py
```

On peut aussi exécuter le fichier directement dans python avec la commande *execfile* :

```
execfile("fichier.py")
```