Initiation à la programmation pour non-informaticiens en Python

Zhentao Li

15 février 2017

Initiation à la programmation pour non-informaticiens en Python

- 11 cours de 2h: CM et TP
- Évaluation: Projet (pas d'examen) pour 3 ECTS.
- Aucune connaissance préalable en programmation n'est requise.
- But: découvrir un outil utile.

Projet

- Créer un programme python (de recherche, loisirs, d'automatisation, etc).
- Doit être approuvé par l'enseignant.
- Réfléchissez-y dès maintenant!
- Le cours sera adapté à vos besoins.

Apprendre à programmer

- Beaucoup de gain pour peu de travail au début.
- Passer du temps à s'habituer aux outils entre les cours.

Contexte

- L'informatique
- Aujourd'hui: Langage de programmation en général.

Langage de programmation

- Pourquoi un autre langage?
  Pour réduire l'ambiguïté.

Langage de programmation

- Pourquoi un autre langage?
  Pour réduire l'ambiguïté.
- *Elle emporte les clefs de la maison au garage.*
  "(clefs de la maison) au garage" ou bien "clefs (de la maison au garage)"

Langage de programmation

- Pourquoi un autre langage?
  Pour réduire l'ambiguïté.
- *Elle emporte les clefs de la maison au garage.*
  "(clefs de la maison) au garage" ou bien "clefs (de la maison au garage)"
- Langage de programmation pour des descriptions et instructions précises.

Langage de programmation

- Pourquoi un autre langage?
  Pour réduire l'ambiguïté.
- *Elle emporte les clefs de la maison au garage.*
  "(clefs de la maison) au garage" ou bien "clefs (de la maison au garage)"
- Langage de programmation pour des descriptions et instructions précises.
- Programme: (fichier) texte contenant des instructions

Langage de programmation

- Pourquoi un autre langage?
  Pour réduire l'ambiguïté.
- *Elle emporte les clefs de la maison au garage.*
  "(clefs de la maison) au garage" ou bien "clefs (de la maison au garage)"
- Langage de programmation pour des descriptions et instructions précises.
- Programme: (fichier) texte contenant des instructions
- Exécutés ligne par ligne, en ordre des priorités des opérations.

Langage de programmation

- Pourquoi un autre langage?
  Pour réduire l'ambiguïté.
- *Elle emporte les clefs de la maison au garage.*
  "(clefs de la maison) au garage" ou bien "clefs (de la maison au garage)"
- Langage de programmation pour des descriptions et instructions précises.
- Programme: (fichier) texte contenant des instructions
- Exécutés ligne par ligne, en ordre des priorités des opérations.
- **Remarque:** Le langage est là pour vous aider à vous exprimer.

Autre différence entre langage de programmation et langages naturels

Un langage de programmation a typiquement

- Moins de vocabulaire (beaucoup d'expressivité même en connaissant une petite partie du langage).
- Syntaxe et grammaire plus rigide (moins de possibilité pour des phrases structurés différemment).
- Plus de ponctuations.

Python (avantages du langage)

- interprété
- lisible
- facile pour débutants.
- styles différents. Ce cours: *impératif*. *Fait ci! Fait ça!*
- introspectif (`dir`, `globals`, `locals`, `hasattr`, `inspect`, `pdb`, etc)
- *modules* inclus

Écriture d'un programme

- But: amener un programme à un état voulu.

Écriture d'un programme

- But: amener un programme à un état voulu.
- Comment? Décrire les étapes à partir d'un état de départ (le vide).

Écriture d'un programme

- But: amener un programme à un état voulu.
- Comment? Décrire les étapes à partir d'un état de départ (le vide).
- La modification de l'état de départ se fait étape par étape.

Écriture d'un programme

- But: amener un programme à un état voulu.
- Comment? Décrire les étapes à partir d'un état de départ (le vide).
- La modification de l'état de départ se fait étape par étape.
- Une telle description est un *programme*.

Écriture d'un programme

- But: amener un programme à un état voulu.
- Comment? Décrire les étapes à partir d'un état de départ (le vide).
- La modification de l'état de départ se fait étape par étape.
- Une telle description est un *programme*.
- D'abord, regardons des objectifs où il y a une seule étape.

Écriture d'un programme

- But: amener un programme à un état voulu.
- Comment? Décrire les étapes à partir d'un état de départ (le vide).
- La modification de l'état de départ se fait étape par étape.
- Une telle description est un *programme*.
- D'abord, regardons des objectifs où il y a une seule étape.
- **Remarque (avancée):** Écrire des instructions étape par étape n'est pas la seule façon de programmer.

Utilisation (console)

Commençons! Lancer l'interpréteur Python.
Après un peu de texte, vous êtes accueilli par l'invité de commande

```
>>>
```

Vous êtes maintenant dans l'interpréteur.

Utilisation (console)

Entrer votre première ligne de python.

```
>>> print "Hello world"
Hello world
>>>
```

Vous pouvez taper la prochain ligne et ainsi de suite.

```
>>> print "Hello world"
Hello world
>>> print 2+3*3
11
```

Pour quitter l'interpréteur, appuyez sur `ctrl-d` (appuyez sur `ctrl` et sans lâcher `ctrl`, appuyez sur `d`). Sous Windows, c'est `ctrl-z`.

Un point pratique: Python 2 et Python 3

Dans ce cours, nous allons utiliser Python 2 (c.-à-d. 2.7, 2.6, 2.5, etc) qui diffère légèrement de Python 3.
En Python 2

```
>>> print "Hello world"
Hello world
```

En Python 3

```
>>> print("Hello world")
Hello world
```

Utilisation (éditeur texte)

Au lieu de lancer l'interpréteur, vous pouvez

- écrire vos lignes directement dans un fichier texte,
- sauvegarder ce fichier (par ex sous le nom `test.py`) et
- le lancer avec `python test.py`.

L'interpréteur exécute le contenu de `test.py` ligne par ligne.
(Se placer au bon endroit d'abord.)
`test.py` est appelé un *script* Python.
Astuce: Recopier les lignes de l'interpréteur vers l'éditeur de texte.

Utilisation (combiné)

Au lieu de lancer l'interpréteur, vous pouvez

- écrire vos lignes directement dans un fichier texte,
- sauvegarder ce fichier (par ex sous le nom `test.py`) et
- le lancer avec `python -i test.py`. **Notez l'ajout du `-i`!**

Cela exécute le contenu de `test.py` **et** vous donne un invité de commende

```
>>>
```

à l'arrêt (qui arrive soit à la fin de la dernière ligne ou lors d'une erreur).

Erreur

Une des première choses que vous allez rencontrer est une erreur, qui est soit de *syntaxe*

```
>>> pas du python
  File "<stdin>", line 1
    pas du python
        ^
SyntaxError: invalid syntax
```

ou *d'exécution*.

```
>>> print 2/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

Corrigez et re-essayez!

Mode d'apprentissage

Quel est le prochain chiffre dans cette suite 1, 4, 9, 16, 25, ?

- Au début: Apprendre par déduction.

Mode d'apprentissage

Quel est le prochain chiffre dans cette suite 1, 4, 9, 16, 25, ?

- Au début: Apprendre par déduction.
- Plus tard: Apprendre en lisant les descriptions. (Voir les liens vers les mémentos de .)

Mode d'apprentissage

Quel est le prochain chiffre dans cette suite 1, 4, 9, 16, 25, ?

- Au début: Apprendre par déduction.
- Plus tard: Apprendre en lisant les descriptions. (Voir les liens vers les mémentos de .)
- Donc générer beaucoup d'erreurs au début.

Mode d'apprentissage

Quel est le prochain chiffre dans cette suite 1, 4, 9, 16, 25, ?

- Au début: Apprendre par déduction.
- Plus tard: Apprendre en lisant les descriptions. (Voir les liens vers les mémentos de .)
- Donc générer beaucoup d'erreurs au début.
- Autre possibilité: modifier des lignes correctes.

Mode d'apprentissage

Quel est le prochain chiffre dans cette suite 1, 4, 9, 16, 25, ?

- Au début: Apprendre par déduction.
- Plus tard: Apprendre en lisant les descriptions. (Voir les liens vers les mémentos de .)
- Donc générer beaucoup d'erreurs au début.
- Autre possibilité: modifier des lignes correctes.
- **Astuce:** Naviguer l'historique dans l'interpréteur avec les flèches.

Opérations arithmétiques

Remarque: L'interpréteur affiche aussi sans `print`.

- Vous pouvez utiliser Python pour de longs calculs.

```
>>> 5*5
25
>>> (3*3) + (4*4)
25
```

- Deuxième exemple: Conversion de 44 degrés Fahrenheit en Celsius

```
>>> (44 - 32)*5/9
6
>>> (44 - 32)*5/9.0
6.666666666666667
```

- résultat tronqué si on divise deux entier
- multiplication explicite (avec *)

Opération modulo

On peut obtenir le reste d'une division avec %.

```
>>> 20 % 3
2
```

Variables

Dans ce deuxième exemple, on aurait bien voulu avoir des variables comme en mathématiques. On peut.

```
>>> degre_f = 44
>>> degre_c = (degre_f - 32) * 5/9.0
>>> degre_c
6.666666666666667
>>> degre_k = degre_c + 273.15
>>> degre_k
279.81666666666666
```

Variables

- Les variables sont des *noms* (que vous choisissez) associé à des *valeur*.
- Écrire son nom dans un expression le remplace par sa valeur *à ce moment*.
- Associer une deuxième *valeur* au même *nom* remplace l'ancienne valeur associée.
- Autre modification de valeurs (plus tard)
- Quelques restrictions s'appliquent aux noms qu'on peut utiliser.
- Pour des *types* plus complexes, la valeur est en fait un emplacement dans la mémoire (mais nous n'avons pas encore vu ce qu'est un type).

Variables (syntaxe)

- L'association (ou l'*assignation*): `nom = expression`.
- Nous verrons que les variables ont beaucoup plus d'utilité qu'une simplement fonction de mémoire comme dans une calculatrice.
- Plus tard, il faudra aussi faire attention à bien choisir les noms de variables pour représenter leur contenu.

Chargement des données

- Remise à zéro des variables au lancement
- Utiliser l'instruction `input()` pour lancer les mêmes instructions sur des entrée différentes.
- `input()` demande à l'utilisateur pour une entrée

Si le fichier `exemple_lecture.py` contient

```
degre_f = input("Entrez le degre en F: ")
degre_c = (degre_f - 32) * 5/9.0
print "En degre celsius, c'est:"
print degre_c
```

Chargement des données: lancement

Au lancement,

```
$ python exemple_lecture.py
Entrez le degre en F: 44
En degre celsius, c'est:
6.66666666667
```

Quelques types

Implicitement nous avons déjà vu les deux types

- `int` pour des entiers (comme `1`, `-11` et `3`)
- `float` pour des nombre à virgule (comme `0.5` et `3.141592653589793`)

Attention: Les `floats` ne sont pas des fractions exactes.

```
>>> 0.1 + 0.1 + 0.1
0.30000000000000004
```

La fonction type

La fonction type permet d'obtenir le type d'une expression.

```
>>> type(3)
<type 'int'>
>>> x = 1/2.0
>>> type(x)
<type 'float'>
>>> type(0.1 + 0.1 + 0.1)
<type 'float'>
```

Cette fonction est peu utilisée hors de l'interpréteur et typiquement utilisée pour examinée le type d'une variable.

Conversion de types

Nous avons déjà vu comment créer des constantes d'un type (par ex 3 et `0.1`). Voici une autre façon.
Nous pouvons appeler la fonction du nom du type (comme `int` et `float`) pour convertir une valeur en ce type.
Convertir le int 9 en un float et le float 3.2 en un int.

```
>>> float(9)
9.0
>>> int(3.2)
3
```

C'est aussi une autre façon d'obtenir une valeur de ce type.

```
>>> x = int()
>>> x
0
```

Le type booléen `bool`

- `bool`: prends deux valeurs `True` et `False` (et les constantes `bool` sont créés ainsi).
- Seront utilisés pour l'exécution conditionnelle d'instructions.
- Beaucoup d'opérations retournent un `bool`, comme les comparateurs

```
>>> 2 > 8
False
>>> 3 == 3
True
>>> x = (5 < 7)
>>> x
True
```

Opérations sur `bool`

Il existe plusieurs opération booléennes dont `or`, `and` et `not`.

```
>>> (2 > 8) or (3 == 3)
True
>>> not (2 > 8)
True
>>> (2 > 8) and x
False
>>> (9 > 24) and inexistant
False
```

Remarque (avancée): Python utiliser une évaluation paresseuse: si le premier élément de `and` évalue à `True` alors, il n'évalue pas le deuxième élément.

Contrôle du flux des instructions

- L'interpréteur Python exécute les instructions ligne par ligne. Mais...
- Nous allons maintenant voir des instructions qui peut guider son exécution.
- Notamment en sautant des instructions ou en répétant des lignes.

```
>>> x = 3
>>> if x < 0:
...     print "x est négatif"
...
>>> x = -1
>>> if x < 0:
...     print "x est négatif"
...
x est négatif
```

- C'est le premier exemple d'une *instruction composée*.

Instructions composées

```
if x <= 0:
    print "x est négatif"
    y = 3
```

- Une instruction composée contient
  - une ligne d'en-tête terminée par deux-points :
  - un **bloc d'instructions** indenté au même niveau.
- S'il y a plusieurs instructions indentées sous la ligne d'en-tête, elles doivent être *exactement* au même niveau (par exemple, un décalage de 4 espaces).
- On peut imbriquer des instructions composées pour réaliser des structures de décision complexes.

Exécution contionnelle `if`, `elif` et `else`

```
x = 0
if x <= 0:
    print "x est négatif"
elif x % 2 == 1:
    print "x est positif et impair"
else:
    print "x n'est pas négatif et est pair"
```

Essayez de changer la valeur de `x` sur la première ligne.

Exécution contionnelle: syntaxe

```
if expression1:
    instruction1
elif expression2:
    instruction2
else:
    instruction3
```

Lorsque nous arrivons sur la ligne `if expression1:`,

- `expression1` est évaluée et si sa valeur est `True` le bloc `instruction1` est exécuté.
- Sinon (`expression1` évaluée à `False`), `expression2` est évaluée et si sa valeur est `True` le bloc `instruction2` est exécuté.
- Sinon (`expression1` et `expression2` sont tous les deux `False`), `instruction3` est évalué.

Remarque (avancée): Des variables de type autre que `bool` peuvent aussi être utilisés comme condition

Boucle `while`

Pour re-exécuter les mêmes lignes plusieurs fois, nous pouvons utiliser l'instruction `while`.

```
>>> x = 257
>>> compte = 0
>>> while x > 1:
...     x = x / 2
...     compte = compte + 1
...
>>> compte
8
```

Cet exemple calcul le log en base 2 de `x` arrondi vers le bas.
Remarque: À la ligne `x = x / 2` nous avons un exemple d'une assignation à `x` d'une valeur obtenue de la valeur de `x` avant l'assignation. C'est un outil très utile.

Syntaxe de `while`

```
while expression:
    instructions
```

- Tant que `expression` évalue à True, le bloc `instructions` est exécuté
- `expression` est reévaluée avant chaque exécution du bloc `instructions`.

Interrompre une boucle

Nous pouvons aussi sortir d'une boucle avec l'instruction `break`.

```
>>> x = 257
>>> compte = 0
>>> while True:
...     x = x / 2
...     compte = compte + 1
...     if x <= 1:
...         break
...
>>> compte
8
```

Cette version est *presque* équivalent à la boucle précédente.
Qu'arrive-t-il si on met `x = 1` en première ligne?

Les chaînes de caractères `str`

Un autre type sont les *chaînes de caractères* `str`.

```
>>> print "Hello world"
Hello world
>>> type("Hello world")
<type 'str'>
>>> "Hello world"
'Hello world'
```

`"Hello world"` est une chaîne de longueur 11.

```
>>> len("Hello world")
11
```

Comme son nom l'indique, ce sont les caractères suivants les unes après les autres `'H'`, `'e'`, `'l'`, `'l'`, `'o'`, `' '`, `'w'`, `'o'`, `'r'`, `'l'`, `'d'`.

Création de chaînes de caractères

Les chaînes de caractères sont un peu plus complexe que les `int`, `float` et `bool` déjà vu pour cette raison.
Voici trois syntaxes différentes pour créer des chaînes de caractères.

```
>>> s1 = "Hello world"
>>> s2 = 'Hello world'
>>> s3 = """Hello world"""
```

Elles peuvent être incluses entre simples quotes (apostrophes) ou doubles quotes (guillemets) ou triple quotes (triple guillemets).

Chaînes de caractères: caractère d'échappement

À priori, nous voulons être capable de représenter tout caractère.
Comment peut-on créer une chaîne avec un seul caractère: " ?

```
>>> '"'
'"'
>>> "\""
'"'
```

\ est un *caractère d'échappement*. Le caractère qui vient après est interprété différament.
(Comment concevoir un tel système de représentation?)

Chaînes de caractères: caractère d'échappement

Quelques combinaisons courantes sont

- \\ donne \
- \n donne une fin de ligne
- \t donne un tab (normalement 4 espaces)

```
>>> print "Ligne 1\nLigne 2\nLigne 3"
Ligne 1
Ligne 2
Ligne 3
>>>
```

Chaînes de caractères: chaînes multi-lignes

La syntaxe à trois guillemets permet aussi d'obtenir des chaînes multi-lignes.

```
>>> s = """Ligne 1
... Ligne 2
... Ligne 3"""
>>> s
'Ligne 1\nLigne 2\nLigne 3'
>>> print s
Ligne 1
Ligne 2
Ligne 3
```

Chaînes de caractères: Opérations courantes

- Concaténation

```
>>> s + s
'Ligne 1\nLigne 2\nLigne 3Ligne 1\nLigne 2\nLigne 3'
```

- Répétition

```
>>> "oysters "*3 + "eat "*3
'oysters oysters oysters eat eat eat '
```

- Élimination d'espaces blancs (espace, fin de ligne, etc)

```
>>> "    ab cd  \n".strip()
'ab cd'
```

Chaînes de caractères: Opérations courantes

- Séparation et recombinaison (interaction avec le type `list`, à voir au prochain cours). Très utile pour charger des données.

```
>>> s = "Des mots separes d'espaces"
>>> liste_de_mots = s.split()
>>> liste_de_mots
['Des', 'mots', 'separes', "d'espaces"]
>>> s_virgule = ",".join(liste_de_mots)
>>> s_virgule
```

- Remplacement

```
>>> s_virgule.replace("d'espaces", "de,virgules")
'Des,mots,separes,de,virgules'
```

- Tests

```
>>> s_virgule.startswith("Des,")
True
```

Chaînes de caractères: Opérations courantes

- Sous-chaîne (nous verrons la syntaxe plus en détails lors de la discussion des `list`)

```
>>> s_virgule[2:21]
"s,mots,separes,d'es"
```

Ici nous voyons aussi un premier exemple d'appel de fonctions.

- `len(s)` (appel `nom_de_fonction(parametre1, parametre2, ...)`)
- `s.replace("a", "b")` (appel `variable.nom_de_fonction(parametre1, parametre2, ...)`)

C'est une syntaxe que nous allons souvent revoir.

Sortie et chaînes formatées

Nous avons déjà utilisé plusieurs fois l'instruction `print`. Voici quelques (autres) forme de son utilisation.

```
>>> a = 2
>>> b = 5
>>> print "La somme de", a, "et", b, "est", a + b
```

On passe ici une liste de 6 arguments à `print` (séparés de virgules):

- "La somme de"
- a
- "et"
- b
- "est"
- a + b

Sortie et chaînes formatées

Deux alternatives plus “modernes”.

```
print "La somme de %s et %s est %s" % (a, b, a + b)
print "La somme de {} et {} est {}".format(a, b, a + b)
```

% est un autre caractère d’échappement pour les chaînes de caractères.

Commentaires

Rappel: Vos programmes seront constamment lu par des humains, incluant vous même lors de sa modification.
Commentaires: texte non-exécuté

```
>>> x = 3 # Ceci est un commentaire
>>> # Ceci est un autre commentaire
```

Ajout d'information supplémentaire lorsque le fonctionnement n'est pas évident.
Tout ce qui est après # sur une ligne est ignoré par l'interpréteur.

Commentaires multi-lignes

Les chaînes de caractères multi-lignes servent aussi de commentaires. Rappel qu'ils ne sont pas affichés s'ils sont exécuté à l'intérieur d'un script.

```
""" Ceci est
un commentaire sur
plusieurs lignes
"""
```