Python
niveau débutant

Vincent Le Gallic

Séminaire Technique du Cr@ns

4 février 2014

(librement inspiré des slides de Pauline Pommeret)

C'est quoi ?

Python est un langage de programmation :

- interprété (exécuté par un interpréteur, par opposition à compilé)
- orienté objet (tout est objet, que l'on définit et manipule)
- impératif (suite d'instructions)
- à typage dynamique (détections des erreurs de type à l'exécution, c'est-à-dire le plus tard possible) fort (peu de conversions implicites)
- équipé d'un garbage collector (qui gère la mémoire à votre place) et d'un système de gestion d'exceptions (affiche un message au lieu de crasher bêtement)
- multi-plateforme (implémentations sous Windows, Linux, Mac OS)

Pourquoi l'utiliser ?

- Relativement simple, souple et facile à apprendre
- Bibliothèque standard très fournie
- Plus succint : *Life is short, you need Python !*[1]
- Dans */usr/scripts/*, on trouve 438 scripts en Python

1. Bruce Eckel, membre du comité de normalisation du C++ et auteur de Thinking in Java

Comment l'utiliser ?

- python en mode « interactif »
  - `python`
  - `ipython`
  - `bpython`
- exécuter un script python
  - `python script.py`
  - exécuter directement `./script.py` si :
    - il est exécutable (`chmod +x script.py`)
    - il commence par un shebang `#!/usr/bin/env python`[2]
    - il est correctement encodé[3]

2. Au Cr@ns, préférer `/usr/scripts/python.sh`, qui fait en sorte qu'on puisse importer facilement les scripts de `/usr/scripts/`.
3. L'encodage du fichier doit correspondre à l'encodage déclaré

Avec quoi je code ?

Avec votre éditeur de texte favori.

- `nano`, `vim`, `emacs`
- SciTE, Kate, Eclipse (avec le plugin PyDev), Code : :Blocks, Python IDE (pour les windowsiens endurcis), Xcode (sous Mac OS)

Attention, Si mon fichier `essai.py` contient

```
print "essai : é"
```

et que je fais naïvement `python essai.py`, j'obtiens

```
SyntaxError: Non-ASCII character '\xc3' in file
a.py on line 1, but no encoding declared; see
http://www.python.org/peps/pep-0263.html for
details
```

Il faut ajouter à la deuxième ligne du fichier :

```
# -*- coding: utf-8 -*-
```

Les types

- En python, tout est objet. Un objet peut être assigné à une variable, passé comme paramètre à une fonction.
- Chaque objet a un type
- Pour le connaître : `type(objet)`
- Python possède de base plein de types dits `built-in`

Types simples

- `int` : entier signé compris entre[4] $-2^{63}+1$ et $2^{63}-1$ (`0, 1, -5`)
- `long` : entier arbitrairement long (`0L, 9223372036854775808L`)
- `float` : nombre à virgule (`0.3, 2e+45`)
- `str` : chaîne de caractères, aussi appelée chaîne d'octets (`"a", "plo\nuf "`)
- `unicode` : chaîne unicode, suite de points de code unicode (`u"a"`) (cf encodages)
- `bool` : booléen, il n'en existe que deux : `True` et `False`
- `NoneType` : un seul objet : `None`. C'est le "rien" de Python. (Il est typiquement renvoyé par une fonction qui ne renvoie pas de résultat.)

4. si vous êtes en architecture 32bits, 63 est à remplacer par 31

Manipulation des chaînes

- `s = "chAinE exEmple"`
- `s.upper()` : `"CHAINE EXEMPLE"`
- `s.lower()` : `"chaine exemple"`
- `s.capitalize()` : `"Chaine exemple"`
- `s.title()` : `"Chaine Exemple"`
- `s = "j'ai une apostrophe"`
- `s = 'je "cite"'`
- `s = "je \"m'élange\"`
- `s = """Je n'ai pas à me soucier des "guillemets" ni des retours à la ligne"""`
- `s.replace("avant", "apres")` : remplace toutes les occurrences
- `len(s)` : longueur de la chaîne

Types composés

- `list` : liste d'objets (pas forcément de même type)
  - déclarer une liste : `l = [3, "a", []]`
  - accéder à un élément : `l[0], l[2], l[-1]`
  - ajouter un élément : `l.append(42)` ou `l.insert(2, "b")`
  - retirer un élément : `del l[1]` ou `l.remove("a")` ou `valeur = l.pop(2)`
  - affecter un élément : `l[0] = 4`
  - concaténer des listes : `l1.extends(l2)`
- `tuple` : n-uplet.
  - déclarer un tuple : `t = (4, "abc")`
  - tuple vide : `()`, tuple à un élément : `(1,)`
  - accéder à un élément : `t[0], t[1]`
- `len(l)` : obtenir la longueur
- NB : pas de "tableau" en python, la liste a le comportement qu'on attend d'un tableau.

Table de hashage

`dict` : dictionnaire, associe une valeur à chaque clé

- déclarer un dictionnaire : `d = {"a" : 26, 5 : []}`
- accéder à une valeur : `d[5]`
- ajouter une clé/remplacer une valeur : `d[5] = "valeur"`
- retirer une valeur : `del d["a"]` ou `valeur = d.pop(5)`
- accéder à une valeur sans échouer : `d.get(6)` (renverra `None` si la clé `6` n'existe pas) ou `d.get(6, "rien")` (renverra `"rien"` si la clé `6` n'existe pas).

Les instructions

- Elles servent à construire le code.
- Ce sont des mots-clés réservés
- Il y en a exactement 26 :

  `as assert break class continue def del elif else except exec finally for from global if import lambda pass print raise return try while with yield`

  (Si on ajoute `and or in is not` (opérateurs booléens), on a les 31 keywords réservés[5] de Python).
- On va voir l'utilité d'un certain nombre d'entre eux.

5. Vous ne pouvez pas les utiliser comme nom de variable

Affichage

print

- `print mon_objet`
- Affiche la représentation sous forme de texte de l'objet
- Appelle la méthode spéciale `mon_objet.__str__()` [6]
- Attention, en python3, `print` n'est plus une instruction et doit être utilisée comme une fonction : `print("a")`

6. cf la programmation orientée objet, séminaire python advanced

Contrôle de flux

En python, l'indentation des blocs est obligatoire et a un sens.

- ```
  if condition1:
      instruction si condition1 est vraie
  elif condition2:
      instruction si condition1 est fausse
      et condition2 est vraie
  else:
      instruction si les deux sont fausses
  ```
- Les blocs `elif` et `else` sont facultatifs
- Les `conditions` peuvent être :
  - un booléen : `True`, `False`
  - une expression dont le calcul donne un booléen : `a < 3`
  - un autre objet : `"plouf"`
    Il est alors **converti** en `bool`.
  - `None`, `False`, `0`, `0L`, `0.0`, `0j`, `""`, `[]`, `()`, `{}` et `set()` sont convertis en `False`, le reste[7] en `True` (par exemple : `"False"`).

7. Sauf un objet défini par l'utilisateur pour se comporter autrement.
  ```

Opérateurs

Opérateurs booléens	
`< / >`	strictement inférieur/supérieur à
`<= / >=`	inférieur/supérieur ou égal à
`==`	égal à
`!=`	différent de
`e in l`	appartient à (`l` doit être itérable)
`o is objet`	est le même objet dans la mémoire
`not / and / or`	non/et/ou booléens [8]

Opérateurs de calcul [9]	
`+`	addition, concaténation
`- / *`	soustraction, division (entière), multiplication
`**`	exponentiation
`%`	modulo

8. Attention si on ne les utilise pas avec des booléens
9. Ces opérateurs appellent la fonction spéciale (par exemple `.__add__()`) du premier sur le deuxième. Cf python advanced.

Les boucles

- ```
  for i in [0, 1, 2, 3, 4, 5, 6]:
      if i == 0:
          continue # passe à l'itération suivante
      elif i == 4:
          break    # interrompt la boucle
      print i,
  ```

  ```
  1 2 3
  ```

- ```
  while condition:
      corps de la boucle
  ```
- si `condition` est fausse, aucune instruction n'est exécutée.
- **`continue`** et **`break`** fonctionnent aussi

Les boucles : astuces

- `enumerate` est une fonction built-in qui prend une liste et retourne la liste des couples (indice, élément).
-
```
for (i, e) in enumerate(l):
    print "L'élément {} est {}".format(i, e)
```
- Les dictionnaire ont une méthode `.iteritems()` qui permet de parcourir les couples (clé, valeur).
-
```
for (k, v) in d.iteritems():
    print "v={}, k={}".format(v, k)
```

Gestion des erreurs

- Une "erreur" en Python s'appelle une exception.
- Les exceptions sont des objets dont le type dépend de la raison de l'erreur.
- Elles héritent toutes de `BaseException`
- Il existe `SystemExit` (l'intépréteur va quitter), `KeyboardInterrupt` (levée en cas de Ctrl+C ou de réception d'un SIGINT). Et enfin `Exception`, dont héritent toutes les "vraies" exceptions.
- On rattrape toujours un **type** d'erreur. Donc ça inclus les exceptions de ce type, mais aussi de tous ceux qui en héritent.

try, except, finally et raise

```
try:
    print 1/a
    print "la division a réussi"
except ZeroDivisionError as e:
    print "a vaut 0, on ne peut pas diviser"
except NameError as e:
    # autre erreur, autre action
    print "la variable a n'existe pas"
    raise ValueError("erreur levée manuellement")
except (TypeError, ValueError) as e:
    # pour faire la même chose pour ces deux erreurs
    print "Erreur de type %s, on s'arrête" % (type(e))
    raise # Relève la même erreur (e)
finally:
    # bloc toujours exécuté, même si il y eu un raise
    print "dans tous les cas j'affiche ça"
```

Fabriquer ses fonctions

- ```
  def mafonction(param1, param2):
      """ Explication sur ma fonction. """
      if un_cas:
          return param1 ** param2 + 42
      elif un_deuxieme_cas:
          return
          # ce code ne sera jamais lu
          print "jamais affiché"
      else:
          print "quelque chose"
  ```
- Sans `return` ou avec `return` mais sans objet, la fonction renverra `None`
  ```

Des fonctions built-in

- On peut les écraser, mais on a intérêt à savoir ce qu'on fait.
- `int, float, str, unicode, list, tuple` : convertissent en le type correspondant (NB : `int("2F", 16)`).
- `enumerate, len` : cf avant
- `open` : pour ouvrir un fichier[10]
- `range(5)` : renvoie la liste `[0, 1, 2, 3, 4]`
- `zip([1, 2, 3, 4], ["a", "b", "c"])` renvoie la liste des paires `[(1, 'a'), (2, 'b'), (3, 'c')]` (s'arrête dès qu'une liste est épuisée)
- `sum(l)` : renvoie la somme des éléments. On peut donner un élément de départ : `sum([[5], [6]], [])` (calcul *in place* donc ne marche pas avec des strings)

10. cf séminaire advanced

- `all(l)` : renvoie `True` si tous les éléments de `l` sont vrais.
- `any(l)` : renvoie `True` si au moins un des éléments de `l` est vrai.
- `bin(x), hex(x), oct(x)` : renvoie la représentation en binaire/hexadécimal/octal du nombre (un string)
- `chr(i), unichr(i) / ord(c)` : renvoie le i-ème caractère de la table ASCII, ou unicode / renvoie le numéro dans la table du caractère (`ord` marche au-delà de 255, pas de uniord).
- `cmp(x, y)` : `-1` si `x < y`, `0` si `x == y`, `1` si `x > y`
- `divmod(a, b)` : renvoie `(quotient, reste)`
- `eval("a = 5 + 3")` : ça marche. Faites vraiment attention. N'utilisez pas ça.
- `filter(fonction_test, l)` : renvoie la liste des objets pour lesquels le test est vrai. Renvoie un `str` ou un `tuple` si c'est le type de `l`.

- `isinstance(o, typ)` : la bonne méthode pour tester le type d'un objet.
  `type(i) == int` c'est mal. `type(i) == type(0)`, c'est pire. `isinstance(i, int)` c'est bien.
  Cas très utile : `isinstance(s, basestring)` (vrai si `s` est de type `unicode` **ou** `str`)
- `issubclass(classe1, classe2)` : teste si un type est un sous-type d'un autre.
  Pour ces deux fonctions, le deuxième paramètre peut être un tuple de classe et le test fera un **ou**.
- `map(f, l1, l2)` : renvoie `[f(l1[0], l2[0]), f(l1[1], l2[1]), ...]`. Complète les listes trop courtes avec des `None`. (on peut donner un nombre quelconque de listes)
- `max(l), min(l)` : renvoie le maximum/minimum d'un itérable. Lève une erreur si il est vide.

Intéraction avec l'utilisateur

- `s = raw_input("Entrez un texte :")` : récupère un string entré par l'utilisateur (le paramètre est optionnel).
- `a = input("Valeur de a :")` : équivalent à `eval(raw_input())`. Attention :
  - L'utilisateur doit entrer une expression Python valide, sous peine de levée d'exception
  - L'utilisateur a alors accès à toute variable de votre programme...

Les modules

- `import module` puis `module.fonction()`
- `from module import fonction` puis `fonction()`. Utile quand on n'en a besoin que d'une ou deux.
- `from module import *` : ne faites **jamais** ça.
- `import module_au_nom_beaucoup_trop_long as mod` puis `mod.fonction()` : raccourci pratique.
- `from module import fonction_au_nom_long as foo` puis `foo()`.
- Si on a importé sans utiliser `from`, on peut faire `reload(module)` pour recharger le module si il a été modifié (si on a utilisé `as mod` il faut appeler `reload(mod)`).

Point sur l'encodage

- Python manipule des chaînes de caractères (type `str`) et des chaînes unicode (type `unicode`)
- `str` $\underset{\text{encodage}}{\overset{\text{décodage}}{\rightleftarrows}}$ `unicode`
- Pour passer de l'un à l'autre, il faut un **encodage**
- `u"àé"` est une chaîne unicode, elle est représentée dans la mémoire comme une liste de points de codes unicode (le 224ème et le 234ème)
- Si on l'**encode** avec l'encodage `UTF-8`, on obtient une chaîne de caractères `"àé"` représentée dans la mémoire par une suite d'octets (195, 169, 195, 160). Avec un autre encodage, ce sera différent.
- Si on **décode** cette chaîne avec l'encodage `ISO8859-1`, on obtient `u"Ã ã©"`. Si vous rencontrez ça, il y a de grandes chances que quelqu'un ait mal géré d'encodage.

Point sur l'encodage

- Python fait des **conversions implicites** : `u"a" + "b"` marchera (et donnera un `unicode`)
- Tant que les chaînes ne sortent pas de la table ASCII, tout va bien, car c'est l'encodage utilisé pour les conversions implicites. Mais `u"a" + "é"` lèvera une **`UnicodeDecodeError`**.
- Utilisez toujours des `unicode` pour travailler
- **Décodez** au plus vite vers l'unicode les chaînes que vous récupérez (depuis l'utilisateur, un fichier, le réseau...) et **encodez** le plus tard possible vers des chaînes d'octets au moment de faire un output (mais n'oubliez pas de le faire).

Un fichier enregistré sur le disque est une suite d'octets placés là par votre éditeur. Python a besoin de savoir dans quel encodage il est écrit pour comprendre ce que vous y avez mis. C'est à ça que lui sert cette ligne

```
# -*- coding: utf-8 -*-
```

Philosophie python, bonne pratiques

- DRY : *Don't Repeat Yourself*. Si vous recopiez des bouts des codes, quelque chose ne va pas.
- Respectez la PEP8
- Commentez votre code, surtout si vous faites quelque chose d'inattendu.
- Mettez des docstrings :

```
def ma_fonction(param):
    """Cette fonction fait ceci avec ``param``"""
    ...
```

- Découpez un code trop long en modules
- N'utilisez pas de variables globales

ipython

- tab-complétion
- `objet?` → donne de l'aide sur l'objet
- bonne gestion de l'historique
- tab-complétion
- `run monscript.py` pour le lancer dans `ipython`
- `!cat plouf` pour exécuter une commande shell depuis `ipython`

Conclusion

- Gardez la doc à portée de main : http ://docs.python.org (attention à la version)
- Dans le séminaire advanced, on parlera de
  - classe d'objets, comment les définir
  - surcharge
  - héritage
  - itérateurs et compréhension
  - plus de modules standards