Projet informatique 2A : Assembleur Python
—
Séance de tutorat 3

État des lieux

À l'issue du livrable 2, vous avez un *parser* fonctionnel permettant :

- de **vérifier** que la **grammaire** de l'assembleur Python est respectée
- de générer un **objet Python**, aussi complet que possible

Objets Python

```c
typedef unsigned int pyobj_type;

struct pyobj_t;
typedef struct pyobj *pyobj_t;


struct pyobj {

  pyobj_type      type;
  unsigned int    refcount;

  union {
    struct {
      pyobj_t       *value;
      int32_t        size;
    }              list;

    struct {
      char          *buffer;
      int            length;
    }             string;

    py_codeblock   *codeblock;

    union {
      int32_t        integer;
      int64_t        integer64;
      double         real;
      struct {
        double         real;
        double         imag;
      }            complex;
    }            number;
  }             py;
};
```

```c
typedef struct {
  int          version_pyvm;
  struct {
    uint32_t    arg_count;
    uint32_t    local_count;
    uint32_t    stack_size;
    uint32_t    flags;
  }           header;
  pyobj_t     parent;
  struct {
    struct {
      uint32_t   magic;
      time_t     timestamp;
      uint32_t   source_size;
    }          header;
    struct {
      pyobj_t     interned;
      pyobj_t     bytecode;
      pyobj_t     consts;
      pyobj_t     names;
      pyobj_t     varnames;
      pyobj_t     freevars;
      pyobj_t     cellvars;
    }        content;
    struct {
      pyobj_t    filename;
      pyobj_t    name;
      uint32_t   firstlineno;
      pyobj_t    lnotab;
    }         trailer;
  }          binary;
} py_codeblock;
```

Objectif du livrable 3

Les objectifs du livrable 3 sont :

- générer le **bytecode** pour la VM Python à partir des instructions de la section `.text`,
- générer `lnotab`,
- **sérialiser** (écrire dans un fichier binaire) l'objet Python.

Génération du bytecode

Section `.text`

```
1  a = 2
2  b = 4
3
4  if a < b :
5      print( a )
6  else :
7      print( b )
```

```
22 .text
23 .line 1
24         LOAD_CONST              0 # 2
25         STORE_NAME              0 # "a"
26 .line 2
27         LOAD_CONST              1 # 4
28         STORE_NAME              1 # "b"
29 .line 4
30         LOAD_NAME               0 # "a"
31         LOAD_NAME               1 # "b"
32         COMPARE_OP              0 # "<"
33         POP_JUMP_IF_FALSE       label_0
34 .line 5
35         LOAD_NAME               0 # "a"
36         PRINT_ITEM
37         PRINT_NEWLINE
38         JUMP_FORWARD            label_1
39 .line 7
40 label_0:
41         LOAD_NAME               1 # "b"
42         PRINT_ITEM
43         PRINT_NEWLINE
44 label_1:
45 coucou :
46         LOAD_CONST              2 # None
47         RETURN_VALUE
```

En Python 2.7, les instructions ont soit :

- **0 paramètre**, et sont codées sur **1 octet**,
  - `PRINT_ITEM` (l.36)
  - `PRINT_NEWLINE` (l.37)
  - `RETURN_VALUE` (l.47)
- **1 paramètre**, et sont codées sur **3 octets**.
  - `LOAD_CONST` (l.24)
  - `COMPARE_OP` (l.32)
  - `JUMP_FORWARD` (l.38)

```
⟨insn⟩ ::= {'insn::0'}
| {'insn::1'} ({'integer::dec'} | {'symbol'})
```

Interlude : les piles de la VM Python

Pour comprendre chaque instruction, il faut savoir que la VM Python travaille avec des **piles** :

- *S* : la pile d'**évaluation**,
  Exemple : pour évaluer `4+2` :
  - empiler `4`
  - empiler `2`
  - empiler `+`
  - exécuter l'opération `+` (au sommet de la pile) qui va :
    dépiler `4`, puis dépiler `2`, faire l'addition et empiler le résultat.
- *C* : la pile d'**appels de fonctions,**
  - contient, pour chaque appel de fonction, l'adresse de retour (entre autres)
- *B* : la pile des **blocs de code**.
  - on y empile et dépile des informations sur des blocs de code en cours d'exécution

Attention

Tout ceci est géré par la VM en interne, pas par vous !

Description d'une instruction (Section B.2, p.114)

Une instruction est décrite par :

- sa mnémonique,
- son éventuel paramètre,
- ses opcodes (deux chiffres hexadécimaux préfixés par `0x` : `Python2` | `Python3`)

Exemple **sans** paramètre :

`BINARY_ADD` Opcodes : `0x17` | `0x17`

Implémente l'opérateur binaire +.

$B \leftarrow B$ $S \leftarrow S_1$+$S_0, S_1, S_2, \ldots, S_{n-1}$

Exemple **avec** paramètre :

`STORE_NAME` `name_ref` Opcodes : `0x5a` | `0x5a`

Implémente `co_names[name_idx]` $\leftarrow S_0$.

$B \leftarrow B$ $S \leftarrow S$

Conversion d'une instruction en bytecode

Instruction à 0 paramètre :

- occupe 1 octet
- → y placer directement l'opcode associé.

Instruction à 1 paramètre :

- occupe 3 octets
- → placer l'opcode associé en première position,
- → placer le paramètre en positions 2 et 3 (codé sur 16 bits en *little endian*)

Mécanisme `EXTENDED_ARGS`

Avec l'instruction `EXTENDED_ARGS`, on peut passer des paramètres de taille supérieure à $64\,\mathrm{kB}$.
→ Pas géré dans ce projet.

Génération de `lnotab`

lnotab

```
22  .text
23  .line 1
24          LOAD_CONST              0 # 2
25          STORE_NAME              0 # "a"
26  .line 2
27          LOAD_CONST              1 # 4
28          STORE_NAME              1 # "b"
29  .line 4
30          LOAD_NAME               0 # "a"
31          LOAD_NAME               1 # "b"
32          COMPARE_OP              0 # "<"
33          POP_JUMP_IF_FALSE       label_0
34  .line 5
35          LOAD_NAME               0 # "a"
36          PRINT_ITEM
37          PRINT_NEWLINE
38          JUMP_FORWARD            label_1
39  .line 7
40  label_0:
41          LOAD_NAME               1 # "b"
42          PRINT_ITEM
43          PRINT_NEWLINE
44  label_1:
45  coucou :
46          LOAD_CONST              2 # None
47          RETURN_VALUE
```

`lnotab` associe un numéro de ligne du code Python et un décalage dans le bytecode :

- À 0 octet, on a la ligne 0,
- 0 octet plus loin, on a la ligne 1,
- 6 octets plus loin, on a la ligne 2,
- 6 octets plus loin, on a la ligne 4,
- 12 octets plus loin, on a la ligne 5,
- 8 octets plus loin, on a la ligne 7.

Codage différentiel dans `lnotab`

- À 0 octet, on a la ligne 0,
- 0 octet plus loin, on a la ligne 1,
- 6 octets plus loin, on a la ligne 2,
- 6 octets plus loin, on a la ligne 4,
- 12 octets plus loin, on a la ligne 5,
- 8 octets plus loin, on a la ligne 7.

Δ bytecode [octets]	numéro ligne	Δ lignes [lignes]
0	0	0
0	1	1
6	2	1
6	4	2
12	5	1
8	7	2

Codage différentiel dans `lnotab`

- À 0 octet, on a la ligne 0,
- 0 octet plus loin, on a la ligne 1,
- 6 octets plus loin, on a la ligne 2,
- 6 octets plus loin, on a la ligne 4,
- 12 octets plus loin, on a la ligne 5,
- 8 octets plus loin, on a la ligne 7.

Δ bytecode [octets]	~~numéro ligne~~	Δ lignes [lignes]
~~0~~	~~0~~	~~0~~
0	~~1~~	1
6	~~2~~	1
6	~~4~~	2
12	~~5~~	1
8	~~7~~	2

`lnotab` final

`0, 1, 6, 1, 6, 2, 12, 1, 8, 2`

… à charger *en binaire* dans le champ `lnotab` de l'objet `py_codeblock`, qui est un objet `pyobj_t` de type "string" - cf slide 16.

Sérialisation de l’objet Python

Objets Python

```c
typedef unsigned int pyobj_type;

struct pyobj_t;
typedef struct pyobj *pyobj_t;


struct pyobj {

  pyobj_type      type;
  unsigned int    refcount;

  union {
    struct {
      pyobj_t       *value;
      int32_t        size;
    }              list;

    struct {
      char          *buffer;
      int            length;
    }              string;

    py_codeblock   *codeblock;

    union {
      int32_t        integer;
      int64_t        integer64;
      double         real;
      struct {
        double         real;
        double         imag;
      }              complex;
    }              number;
  }             py;
};
```

```c
typedef struct {
  int          version_pyvm;
  struct {
    uint32_t    arg_count;
    uint32_t    local_count;
    uint32_t    stack_size;
    uint32_t    flags;
  }           header;
  pyobj_t      parent;
  struct {
    struct {
      uint32_t   magic;
      time_t     timestamp;
      uint32_t   source_size;
    }          header;
    struct {
      pyobj_t     interned;
      pyobj_t     bytecode;
      pyobj_t     consts;
      pyobj_t     names;
      pyobj_t     varnames;
      pyobj_t     freevars;
      pyobj_t     cellvars;
    }          content;
    struct {
      pyobj_t    filename;
      pyobj_t    name;
      uint32_t   firstlineno;
      pyobj_t    lnotab;
    }          trailer;
  }           binary;
} py_codeblock;
```

Sérialisation

Le but de la sérialisation est de générer un fichier **binaire** à partir de l'objet Python.

Le fichier binaire généré, au format `.pyc`, pourra être exécuté par la VM Python.

```
$ ./python2.7 test.pyc
```

En-tête du fichier binaire

Un fichier `.pyc` comporte toujours un en-tête, avant la sérialisation de l'objet Python (p.84)

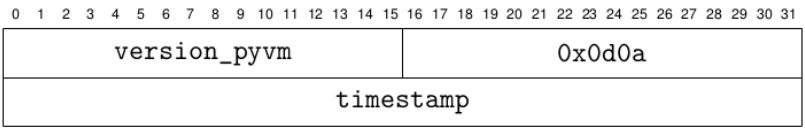


FIGURE 9.3 – En-tête d'un fichier `.pyc` pour Python 2.7.

La valeur de `version_pyvm` (p.85) est à récupérer dans un fichier préalablement compilé ou dans le code assembleur.

Sérialisation de l'objet de code Python (Annexe C p.141)

en-tête de l'objet Python

```
entier arg_count
entier local_count
entier stack_size
entier flags
```

corps de l'objet Python

`STRING` instructions,
`tuple` constantes (`.consts`),
`tuple` noms de symboles (`.names`),
`tuple` noms de variables (`.varnames`),
`tuple` noms de variables libres (`.freevars`),
`tuple` cellules (`.cellvars`),

fin de l'objet Python

`STRING` nom du fichier source Python (`filename`),
`STRING` nom du code (`name`),
`entier` entier donnant le numéro de la première ligne du code (`firstlineno`, souvent 1 pour nous, sauf au livrable 4),
`STRING` `lnotab`

Sérialisation des différents types (Annexe C p.141)

Chaque **type** a un **marqueur** associé (table C.1 p.146)

- Constantes (`null`, `True`, `False`, `None`)
  - Codées par leur marqueur (`'0'`, `'T'`, `'F'`, `'N'`) seulement
- Nombres (voir C.2.2 p.141)
  - entier `INT` : 4 octets en *little endian*
  - réel `FLOAT` ou `BINARY_FLOAT`: 2 possibilités, prendre la plus courte.
  - complexe `COMPLEX` ou `BINARY_COMPLEX` : 2 possibilités
- chaîne de caractères - page suivante.
- liste `LIST` : nombre d'éléments sur 4 octets puis les éléments
- tuple `TUPLE` : nombre d'éléments sur 4 octets puis les éléments

Table C1 (p.146)

Type	Marqueur	Observation
NULL	'0'	Absence d'objet
NONE	'N'	None
FALSE	'F'	False
TRUE	'T'	True
INT	'i'	Entier signé sur 4 octets
INT64	'I'	Entier signé sur 8 octets, plus généré
FLOAT	'f'	Chaîne d'un réel (max : 17 caractères)
BINARY_FLOAT	'g'	Réel binaire sur 8 octets (double)
COMPLEX	'x'	Deux chaînes pour un complexe
BINARY_COMPLEX	'y'	Deux réels pour un complexe
STRING	's'	Chaîne
REF	'r'	Référence à un objet
STRINGREF	'R'	Référence à une chaîne internée
TUPLE	'('	Tuple
SMALL_TUPLE	')'	Petit tuple
LIST	'['	Liste
DICT	'{'	Dictionnaire
SET	'<'	Ensemble
ASCII	'a'	Chaîne ASCII → Unicode
ASCII_INTERNED	'A'	*idem* mais à interner
SHORT_ASCII	'z'	Petite chaîne ASCII → Unicode
SHORT_ASCII_INTERNED	'Z'	*idem*, à interner
CODE	'c'	Objet de code
STOP_ITER	'S'	Arrêt d'un itérateur
ELLIPSIS	'.'	...
LONG	'l'	Entier signé en base 15
UNICODE	'u'	Chaîne Unicode
INTERNED	't'	*idem*, à interner
UNKNOWN	'?'	Objet de type inconnu
FROZENSET	'>'	Ensemble en lecture seule

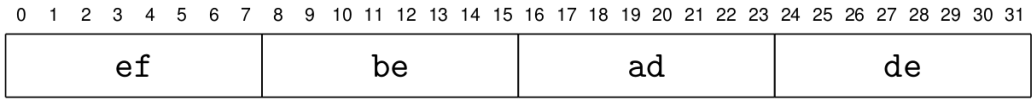
Sérialisation des différents types (Annexe C p.141)

- Sérialisation des chaînes de caractères (section C.2.3 p.142)
  - Les marqueurs possibles pour sérialiser une chaîne sont (table C.1 p.146) `STRING`, `ASCII`, `SHORT_ASCII`, `UNICODE`, `ASCII_INTERNED`, `SHORT_ASCII_INTERNED`.
  - Le support des caractères autres que ASCII (accents, etc) est optionnel.
  - On pourra dans un premier temps utiliser les marqueurs `ASCII`, `SHORT_ASCII` - et leur versions “interned” pour les chaînes intérnées.
  - Puis améliorer en analysant les fichiers binaires produits par le compilateur python2 (*reverse engineering*)
- Sérialisation du `bytecode` et de `lnotab` (p.143-144)
  - Le `bytecode` et `lnotab` sont construits en mémoire dans un objet python, au moyen du champ `string` de l’union du type `pyobj_t`.
  - Mais, dans ce cas, le tableau `char *buffer` du champ `string` ne doit bien sûr pas être entendu comme stockant une “chaîne de caractères”, mais une série d’octets - type `char` entendu comme “entier sur un octet”.
  - Sérialisation : très simple avec la fonction C `fwrite` !

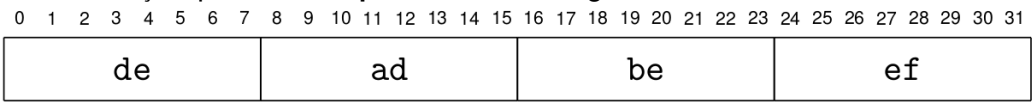
Endianness ou boutisme

Comment stocker une valeur de 32 bits (par exemple `0xdeadbeef`) en mémoire ?

- En commençant par l'octet de **poids faible** ! L'ordre petit-boutiste !



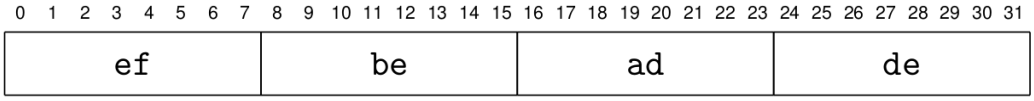
- En commençant par l'octet de **poids fort** ! L'ordre gros-boutiste !



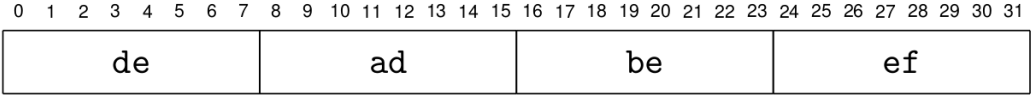
Endianness ou boutisme

Comment stocker une valeur de 32 bits (par exemple `0xdeadbeef`) en mémoire ?

- En commençant par l'octet de **poids faible** ! L'ordre petit-boutiste !



- En commençant par l'octet de **poids fort** ! L'ordre gros-boutiste !



Boutisme de la VM Python

La VM Python est **petit-boutiste**, et votre PC aussi.

- Petit-boutistes célèbres : x86, x86-64,
- Grand-boutistes célèbres : Motorola 6800, Atmel AVR32, OpenRISC,
- Bi-boutistes (!!!) : ARM, RISC-V

Étiquettes / labels

Afin de rendre le code assembleur plus lisible, des **étiquettes** (ou labels) sont utilisées.

Une étiquette sert à **nommer** une **adresse** dans le code.
Plus précisément, c'est un numéro d'octet dans le *bytecode*.

Elles sont utilisées pour :

- les instructions de saut relatif,
- les instructions de saut absolu,
- les instructions relatives aux boucles et exceptions

Un code assembleur avec étiquettes requiert donc une passe d'analyse supplémentaire pour les **remplacer** par les adresses ou intervalles correspondants : construction d'un dictionnaire dont les clés sont les étiquettes et les valeurs leur position dans le bytecode.

Codes de test : option `--easy`

```
$ python2 -m compileall test.py
$ pyc-objdump disasm test.pyc
```

```
.text
.line 1
        LOAD_CONST              0
        STORE_NAME              0
.line 2
        LOAD_CONST              1
        STORE_NAME              1
.line 3
        LOAD_NAME               0
        LOAD_NAME               1
        COMPARE_OP              0
        POP_JUMP_IF_FALSE       label_0
.line 4
        LOAD_NAME               0
        PRINT_ITEM
        PRINT_NEWLINE
        JUMP_FORWARD            label_1
.line 6
label_0:
        LOAD_NAME               1
        PRINT_ITEM
        PRINT_NEWLINE
label_1:
        LOAD_CONST              2
        RETURN_VALUE
```

```
$ python2 -m compileall test.py
$ pyc-objdump disasm test.pyc --easy
```

```
.text
.line 1
        LOAD_CONST              0
        STORE_NAME              0
.line 2
        LOAD_CONST              1
        STORE_NAME              1
.line 3
        LOAD_NAME               0
        LOAD_NAME               1
        COMPARE_OP              0
        POP_JUMP_IF_FALSE       32
.line 4
        LOAD_NAME               0
        PRINT_ITEM
        PRINT_NEWLINE
        JUMP_FORWARD            5
.line 6
        LOAD_NAME               1
        PRINT_ITEM
        PRINT_NEWLINE
        LOAD_CONST              2
        RETURN_VALUE
```

Outillage

Outillage pour cet incrément

- Ecrire des (petits) programmes python2
- Compiler un tel programme avec :
  `python2 -m compileall momprogrammepython2.py`
  => *génère un fichier binaire* `momprogrammepython2.pyc`.
- **Afficher octet par octet le contenu d'un fichier binaire** `.pyc` **avec :**
  `hexdump -C pomprogrammepython2.pyc`
  => *adopter une démarche de reverse engineering pour comprendre le binaire généré par le compilateur python.*
- Générer le code assembleur avec `pyc-objdump` :
  `pyc-objdump disasm pomprogrammepython2.pyc > momprogrammepython2.pys`
  => *génère le fichier assembleur* `momprogrammepython2.pys`
  Voir Annexe D p.149 pour l'installation de `pyc-objdump` *sur la machine virtuelle.*
- Récréer un fichier binaire `.pyc` avec votre programme et comparer le contenu des deux fichiers `.pyc` (au moyen de `hexdump -C` par exemple).

Exemple

Le code python de cet exemple :

```
$ cat test.py
```

```
a = 2
b = 4
if a < b :
        print( a )
else :
        print( b )
```

Exemple

```
$ python2 -m compileall test.py
$ pyc-objdump disasm test.pyc
```

```
(...)
.text
.line 1
LOAD_CONST              0
STORE_NAME              0
.line 2
LOAD_CONST              1
STORE_NAME              1
.line 3
LOAD_NAME               0
LOAD_NAME               1
COMPARE_OP              0
POP_JUMP_IF_FALSE       32
.line 4
LOAD_NAME               0
PRINT_ITEM
PRINT_NEWLINE
JUMP_FORWARD            5
.line 6
LOAD_NAME               1
PRINT_ITEM
PRINT_NEWLINE
LOAD_CONST              2
RETURN_VALUE
```

```
$ hexdump -C test.pyc
```

```
00000000  03 f3 0d 0a 0d e9 94 61  63 00 00 00 00 00 00 00  |.......ac.......|
00000010  00 02 00 00 00 40 00 00  00 73 29 00 00 00 64 00  |.....@...s)...d.|
00000020  00 5a 00 00 64 01 00 5a  01 00 65 00 00 65 01 00  |.Z..d..Z..e..e..|
00000030  6b 00 00 72 20 00 65 00  00 47 48 6e 05 00 65 01  |k..r .e..GHn..e.|
00000040  00 47 48 64 02 00 53 28  03 00 00 00 69 02 00 00  |.GHd..S(....i...|
00000050  00 69 04 00 00 00 4e 28  02 00 00 00 74 01 00 00  |.i....N(....t...|
00000060  00 61 74 01 00 00 00 62  28 00 00 00 00 28 00 00  |.at....b(....(..|
00000070  00 00 28 00 00 00 00 73  09 00 00 00 73 63 72 69  |..(....s....scri|
00000080  70 74 2e 70 79 74 08 00  00 00 3c 6d 6f 64 75 6c  |pt.pyt....<modul|
00000090  65 3e 01 00 00 00 73 08  00 00 00 06 01 06 01 0c  |e>....s.........|
000000a0  01 08 02                                          |...|
000000a3
```