NSI en première (2019-2020)
Fonctionnement 2

Une machine ne comprend que le langage binaire. Un programme, écrit par un humain dans un langage de programmation qu'il comprend, est traduit par un compilateur ou un assembleur, et permet de donner des instructions à la machine dans son langage.

Notions sur le langage machine

Avec le module `dis`, nous pouvons avoir une idée des instructions passées à la machine lorsque nous écrivons un code en Python.

```
import dis
dis.dis('x = 1; x = x + 2')
```

Affichage du code machine désassemblé :

```
  1           0 LOAD_CONST               0 (1)
              2 STORE_NAME               0 (x)
              4 LOAD_NAME                0 (x)
              6 LOAD_CONST               1 (2)
              8 BINARY_ADD
             10 STORE_NAME               0 (x)
             12 LOAD_CONST               2 (None)
             14 RETURN_VALUE
```

Nous suivons ces instructions dans l'ordre :

- le nombre 1 est copié dans un registre ;
- le contenu du registre est copié dans la mémoire à la case adressée par x ;
- la valeur de x est copiée dans un registre ;
- le nombre 2 est copié dans un registre ;
- l'addition des deux est exécutée ;
- le résultat est copié dans la mémoire à la case adressée par x
- la valeur `None` est copiée dans un registre ;
- elle est renvoyée.

Chaque instruction machine est composée d'un nombre en binaire codant l'opération à réaliser, suivi des opérandes, c'est-à-dire les données sur lesquelles l'opération est effectuée, codées également en binaire. Les instructions LOAD et STORE permettent de copier et d'échanger des données entre le processeur et la mémoire.

Par exemple, pour l'instruction `LOAD_CONST`, le code qui correspond à ce que la machine doit exécuter est le nombre 100 en décimal. Ce code est écrit en binaire, soit 01100100 sur un octet.

Vérifions le code de `LOAD_CONST` :

```
>>> dis.opmap['LOAD_CONST']
100
>>> dis.opname[100]
'LOAD_CONST'
```

Dans un langage assembleur, on obtiendrait la suite d'instructions :

```
MOV R0, #1
STR R0, x
LDR R0, x
MOV R1, #2
ADD R0, R0, R1
STR R0, x
```

Ce code est comparable au code figurant en haut de la page.

Dans le code précédent, les instructions sont exécutées dans l'ordre d'écriture, l'une après l'autre. Il est possible de rompre cette séquence, par exemple dans le calcul d'une expression logique ou dans un programme avec un branchement.

Considérons l'expression logique `True and False`.

Dans le cas de cette expression logique, nous voyons bien le fonctionnement de l'opérateur `and` avec le saut à la dernière ligne (`JUMP_IF_FALSE_OR_POP 6`) si la première valeur est `False` et sinon, le renvoi de la deuxième valeur.

```
>>> dis.dis('True and False')
  1           0 LOAD_CONST               0 (True)
              2 JUMP_IF_FALSE_OR_POP     6
              4 LOAD_CONST               1 (False)
        >>    6 RETURN_VALUE
```

Voici un programme avec un branchement dont le code est écrit sur plusieurs lignes entre des triples guillemets :

```
code = """
x = 3
if x < 0:
    y = -x
else:
    y = x
"""
dis.dis(code)
```

Et voici l'affichage obtenu :

```
  2           0 LOAD_CONST               0 (3)
              2 STORE_NAME               0 (x)

  3           4 LOAD_NAME                0 (x)
              6 LOAD_CONST               1 (0)
              8 COMPARE_OP               0 (<)
             10 POP_JUMP_IF_FALSE       20
```

```
4           12 LOAD_NAME                0 (x)
            14 UNARY_NEGATIVE
            16 STORE_NAME               1 (y)
            18 JUMP_FORWARD             4 (to 24)

6       >>  20 LOAD_NAME                0 (x)
            22 STORE_NAME               1 (y)
        >>  24 LOAD_CONST               2 (None)
            26 RETURN_VALUE
```

L'instruction `POP_JUMP_IF_FALSE` suivie de 20 signifie que si le test est faux le programme saute à l'instruction `20 LOAD_NAME` pour charger la valeur de x et la stocker dans y. Sinon, le programme se poursuit avec `12 LOAD_NAME` pour charger la valeur de x, puis avec `14 UNARY_NEGATIVE` prendre son opposée et la stocker dans y avec `16 STORE_NAME` et finalement sauter à la fin du programme ligne 24 avec `JUMP_FORWARD`.

Le module `dis` de Python nous propose d'autres fonctionnalités. Par exemple :

```
>>> def f():
        x = 2
        x = x + 3

>>> code = f.__code__.co_code
>>> for octet in code: # code est du type bytes
        print(octet, end=" ")

100 1 125 0 124 0 100 2 23 0 125 0 100 0 83 0
```

Nous reconnaissons les codes décimaux 100 pour `LOAD_CONST`, 125 pour `STORE_FAST`, etc.
Voici un dernier exemple d'utilisation :

```
>>> def f():
        x, y, z = 1, 2, 3
        t = x * (y + z)

>>> f.__code__.co_consts
(None, 1, 2, 3, (1, 2, 3))
>>> f.__code__.co_varnames
('x', 'y', 'z', 't')
>>> f.__code__.co_stacksize
3
```

La dernière ligne nous renseigne sur la taille de la pile (stacksize) qui est 3. Une pile est une structure de données, on peut l'imaginer comme une pile d'assiettes. Python utilise donc une pile pour "empiler" les données et effectuer le calcul `x*(y+z)`.