

# Python Bytecode

## Python Behind the Scenes

Matías Bordese

PyCon Argentina  
Octubre de 2011



# Python Bytecode

- 1 De .py a .pyc
- 2 Análisis pyc to gráfico
- 3 Experimentos
- 4 Buscando protección



# Python Bytecode

- 1 De .py a .pyc
- 2 Análisis pyc to gráfico
- 3 Experimentos
- 4 Buscando protección



- Código fuente.
- Es human readable.
- Lo podemos correr en cualquier plataforma Python.



- El código Python se compila a una representación interna en bytes (o bytecode) que el intérprete luego ejecuta.
- Esta compilación se cachea en los archivos `.pyc`.
- El intérprete se encarga de ejecutar el código máquina correspondiente a cada bytecode.



# Python Bytecode

- 1 De .py a .pyc
- 2 **Análisis pyc to gráfico**
- 3 Experimentos
- 4 Buscando protección



## Compilando

### .pyc

- Formato estándar de serialización.
- Creado al compilar .py (compile, import).
- No documentado.

### .pyo

- Misma estructura que .pyc.
- -o remueve asserts.
- -oo remueve documentación inline.

### .pyd

- Creado por script freeze.py.
- Objeto compartido (similar DLL, .so) que contiene los objetos Python serializados.
- Ver Tools/freeze en Python source.

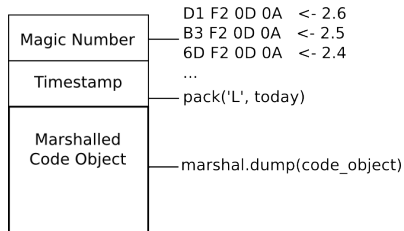


## Estructura de archivos .pyc

### Archivo .pyc

Es un archivo binario que contiene:

- Un número mágico que identifica la versión de Python (4 bytes).
- Un timestamp (4 bytes).
- Un code object serializado con marshal.





## Ejemplo

```
# hello.py  
print "Hello world!"
```

[hello.pyc]

0000:0000	b3 f2 0d 0a 6e 87 99 4a 63 00 00 00 00 00 00 00	..n..Jc.....
0000:0010	00 01 00 00 00 40 00 00 00 73 09 00 00 00 64 00	.....@...s...d.
0000:0020	00 47 48 64 01 00 53 28 02 00 00 00 73 0c 00 00	.GHd..S(...s...
0000:0030	00 48 65 6c 6c 6f 20 77 6f 72 6c 64 21 4e 28 00	.Hello world!N(.
0000:0040	00 00 00 28 00 00 00 00 28 00 00 00 00 28 00 00	...{...{...{...
0000:0050	00 00 73 08 00 00 00 68 65 6c 6c 6f 2e 70 79 73	..s....hello.pys
0000:0060	08 00 00 00 3c 6d 6f 64 75 6c 65 3e 01 00 00 00	....<module>....
0000:0070	73 00 00 00 00	s....

```
>>> import imp, time, struct  
  
>>> imp.get_magic().encode('hex')  
'b3f20d0a'  
  
>>> time.localtime(struct.unpack('L', '\x6e\x87\x99\x4a')[0])  
(2009, 8, 29, 16, 54, 22, 5, 241, 0)
```



## import marshal

- Módulo para escribir y leer objetos Python en formato binario.
- Independiente de la arquitectura.
- Dependiente de la versión de Python.
- Los detalles del formato no están documentados (a propósito!).
- Existe principalmente para leer y escribir code objects en archivos .pyc.

```
marshal.dump(value, fdesc[, version])  
marshal.dumps(value[, version])  
  
marshal.load(fdesc)  
marshal.loads(string)
```



## Code object

- Tipo interno
- Representa el bytecode del código Python
- No tiene un contexto
- Es inmutable y no contiene referencias a objetos mutables
- Se puede ejecutar con `exec` o evaluar mediante `eval`
- Algunos atributos (sólo lectura):
  - `co_code`: string que representa la secuencia de bytecode de las instrucciones
  - `co_consts`: tupla que contiene los literales usados por el bytecode
  - Otros referidos a variables, argumentos y stack



## Ejemplo

```
>>> import marshal, time, struct
>>> f = open("hello.pyc", "rb")
>>> magic = f.read(4)
>>> timestamp = f.read(4)
>>> code = marshal.load(f)

>>> magic.encode('hex')
'b3f20d0a'

>>> time.localtime(struct.unpack('L', timestamp)[0])
(2009, 8, 29, 16, 54, 22, 5, 241, 0)

>>> code
<code object <module> at 0xb7a17698, file "hello.py", line 1>

>>> code.co_code.encode('hex')
'640000474864010053'

>>> code.co_consts
('Hello world!', None)

>>> exec code
Hello world!
```



## import dis

- Módulo para analizar y desensamblar Python bytecode.
- Define el assembler de Python.
  - Los opcodes están listados en `Include/opcode.h`.
  - Python 2.6 tiene 113 opcodes.
  - Cada instrucción consiste de 1-byte + (si requiere) un argumento (16 bits)
  - Soporta argumentos extendidos.
  - Los datos no son parte del bytecode.



## import dis

Algunos opcodes

- `LOAD_CONST(consti)`  
Pushea `co_consts[consti]` en el stack.
- `BUILD_TUPLE(count)`  
Crea una tupla consumiendo `count` items del stack; pushea la tupla resultante.
- `JUMP_IF_TRUE(delta)`  
Si `TOS1` evalúa a `True`, incremente el bytecode counter en `delta`; `TOS` queda en el stack.
- `CALL_FUNCTION(argc)2`  
Llama a una función; `argc` indica el número de parámetros posicionales (low byte) y keyword (high byte); los parámetros y el objeto función se toman del stack; se pushea el valor de retorno.

---

<sup>1</sup>top-of-stack

<sup>2</sup>El resto descriptos aquí:

<http://docs.python.org/library/dis.html#python-bytecode-instructions>



## Ejemplo

```
>>> import dis
>>> dis.disassemble(code)
1          0 LOAD_CONST           0 ('Hello world!')
          3 PRINT_ITEM
          4 PRINT_NEWLINE
          5 LOAD_CONST           1 (None)
          8 RETURN_VALUE
```

- Este es un ejemplo muy simple, con un único code object en el flujo de instrucciones.
- Un módulo que defina clases y funciones es más complejo:
  - Las clases y funciones son de por sí code objects, que se suman a la tupla de consts, anidándose siguiendo la estructura original del módulo.



# Python Bytecode

- 1 De .py a .pyc
- 2 Análisis pyc to gráfico
- 3 Experimentos**
- 4 Buscando protección

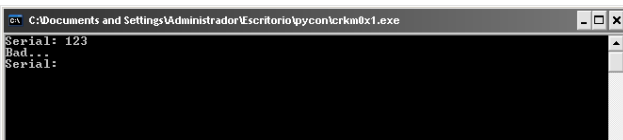




## Caso de estudio

Ingeniería inversa sobre un crackme

- **URL:**  
[http://crackmes.de/users/qfqe/crackme\\_0x01\\_by\\_qfqe/](http://crackmes.de/users/qfqe/crackme_0x01_by_qfqe/)
- **Autor:** qfqe
- El objetivo es encontrar el serial válido.



```
CA C:\Documents and Settings\Administrador\Escritorio\pycon\crkm0x1.exe
Serial: 123
Bad...!
Serial:
```

ejecutable  
py2exe



# unpy2exe

Obtener los pyc de un py2exe

- **URL:** <https://github.com/matiasb/unpy2exe>
- **Licencia:** GPL
- **Autor:** Matías Bordese
- **Versión:** 0.1
- Permite obtener los archivos `.pyc` de un exe compilado con py2exe<sup>3</sup>.



<sup>3</sup><http://www.py2exe.org/>



## Interpretando bytecode

```
34  LOAD_FAST          0 (inputSerial)
37  LOAD_CONST         2 ('')
40  LOAD_ATTR          2 (join)
43  LOAD_GLOBAL        3 (map)
46  LOAD_CONST         3 (<code object <lambda>>)
49  MAKE_FUNCTION      0
52  LOAD_CONST         4 (225)
55  LOAD_CONST         5 (245)
58  LOAD_CONST         6 (241)
61  LOAD_CONST         7 (230)
64  LOAD_CONST         8 (215)
67  LOAD_CONST         9 (161)
70  LOAD_CONST        10 (202)
73  LOAD_CONST        11 (200)
-->76 BUILD_LIST         8
79  CALL_FUNCTION       2
82  CALL_FUNCTION       1
85  COMPARE_OP         2 (==)
88  JUMP_IF_FALSE     10 (to 101)
91  POP_TOP
92  LOAD_CONST        12 ('Good!')
...
```

200
202
161
215
230
241
245
225
lambda
map
".join
inputSerial



## Interpretando bytecode

```
34  LOAD_FAST          0 (inputSerial)
37  LOAD_CONST         2 ('')
40  LOAD_ATTR          2 (join)
43  LOAD_GLOBAL         3 (map)
46  LOAD_CONST         3 (<code object <lambda>>)
49  MAKE_FUNCTION      0
52  LOAD_CONST         4 (225)
55  LOAD_CONST         5 (245)
58  LOAD_CONST         6 (241)
61  LOAD_CONST         7 (230)
64  LOAD_CONST         8 (215)
67  LOAD_CONST         9 (161)
70  LOAD_CONST        10 (202)
73  LOAD_CONST        11 (200)
-->76  BUILD_LIST         8
79  CALL_FUNCTION       2
82  CALL_FUNCTION       1
85  COMPARE_OP         2 (==)
88  JUMP_IF_FALSE      10 (to 101)
91  POP_TOP
92  LOAD_CONST        12 ('Good!')
...
```

200

202

161

215

230

241

245

225

lambda

map

".join

inputSerial



## Interpretando bytecode

```
34  LOAD_FAST          0 (inputSerial)
37  LOAD_CONST         2 (')
40  LOAD_ATTR          2 (join)
43  LOAD_GLOBAL        3 (map)
46  LOAD_CONST         3 (<code object <lambda>>)
49  MAKE_FUNCTION      0
52  LOAD_CONST         4 (225)
55  LOAD_CONST         5 (245)
58  LOAD_CONST         6 (241)
61  LOAD_CONST         7 (230)
64  LOAD_CONST         8 (215)
67  LOAD_CONST         9 (161)
70  LOAD_CONST        10 (202)
73  LOAD_CONST        11 (200)
76  BUILD_LIST         8
-->79  CALL_FUNCTION      2
82  CALL_FUNCTION      1
85  COMPARE_OP         2 (==)
88  JUMP_IF_FALSE     10 (to 101)
91  POP_TOP
92  LOAD_CONST        12 ('Good!')
...
```

[225, ..., 200]

lambda

map

".join

inputSerial



## Interpretando bytecode

```
34  LOAD_FAST          0 (inputSerial)
37  LOAD_CONST         2 (')
40  LOAD_ATTR          2 (join)
43  LOAD_GLOBAL         3 (map)
46  LOAD_CONST         3 (<code object <lambda>>)
49  MAKE_FUNCTION      0
52  LOAD_CONST         4 (225)
55  LOAD_CONST         5 (245)
58  LOAD_CONST         6 (241)
61  LOAD_CONST         7 (230)
64  LOAD_CONST         8 (215)
67  LOAD_CONST         9 (161)
70  LOAD_CONST        10 (202)
73  LOAD_CONST        11 (200)
76  BUILD_LIST         8
-->79  CALL_FUNCTION       2
82  CALL_FUNCTION       1
85  COMPARE_OP         2 (==)
88  JUMP_IF_FALSE      10 (to 101)
91  POP_TOP
92  LOAD_CONST        12 ('Good!')
...
```

[225, ..., 200]

lambda

map

".join

inputSerial



# Interpretando bytecode

Qué hace el lambda?

```
0  LOAD_GLOBAL      0  (chr)
3  LOAD_FAST        0  (x)
6  LOAD_CONST       0  (144)
9  BINARY_XOR
10 CALL_FUNCTION    1
13 RETURN_VALUE
```



# Interpretando bytecode

## Claramente

0	LOAD_GLOBAL	0 ( <b>chr</b> )
3	LOAD_FAST	0 ( <b>x</b> )
6	LOAD_CONST	0 (144)
9	BINARY_XOR	
10	CALL_FUNCTION	1
13	RETURN_VALUE	

```
lambda x: chr(x ^ 144)
```





## Interpretando bytecode

```
34  LOAD_FAST          0 (inputSerial)
37  LOAD_CONST         2 (')
40  LOAD_ATTR          2 (join)
43  LOAD_GLOBAL        3 (map)
46  LOAD_CONST         3 (<code object <lambda>>)
49  MAKE_FUNCTION      0
52  LOAD_CONST         4 (225)
55  LOAD_CONST         5 (245)
58  LOAD_CONST         6 (241)
61  LOAD_CONST         7 (230)
64  LOAD_CONST         8 (215)
67  LOAD_CONST         9 (161)
70  LOAD_CONST        10 (202)
73  LOAD_CONST        11 (200)
76  BUILD_LIST         8
79  CALL_FUNCTION      2
-->82 CALL_FUNCTION      1
85  COMPARE_OP         2 (==)
88  JUMP_IF_FALSE     10 (to 101)
91  POP_TOP
92  LOAD_CONST        12 ('Good!')
...
```

map(lambda, list)

".join

inputSerial



## Interpretando bytecode

```
34  LOAD_FAST          0 (inputSerial)
37  LOAD_CONST         2 (')
40  LOAD_ATTR          2 (join)
43  LOAD_GLOBAL        3 (map)
46  LOAD_CONST         3 (<code object <lambda>>)
49  MAKE_FUNCTION      0
52  LOAD_CONST         4 (225)
55  LOAD_CONST         5 (245)
58  LOAD_CONST         6 (241)
61  LOAD_CONST         7 (230)
64  LOAD_CONST         8 (215)
67  LOAD_CONST         9 (161)
70  LOAD_CONST        10 (202)
73  LOAD_CONST        11 (200)
76  BUILD_LIST         8
79  CALL_FUNCTION       2
-->82 CALL_FUNCTION       1
85  COMPARE_OP         2 (==)
88  JUMP_IF_FALSE      10 (to 101)
91  POP_TOP
92  LOAD_CONST        12 ('Good!')
...
```

map(lambda, list)

".join

inputSerial



## Interpretando bytecode

```
34  LOAD_FAST          0 (inputSerial)
37  LOAD_CONST         2 (')
40  LOAD_ATTR          2 (join)
43  LOAD_GLOBAL        3 (map)
46  LOAD_CONST         3 (<code object <lambda>>)
49  MAKE_FUNCTION      0
52  LOAD_CONST         4 (225)
55  LOAD_CONST         5 (245)
58  LOAD_CONST         6 (241)
61  LOAD_CONST         7 (230)
64  LOAD_CONST         8 (215)
67  LOAD_CONST         9 (161)
70  LOAD_CONST        10 (202)
73  LOAD_CONST        11 (200)
76  BUILD_LIST         8
79  CALL_FUNCTION       2
82  CALL_FUNCTION       1
-->85  COMPARE_OP         2 (==)
88  JUMP_IF_FALSE      10 (to 101)
91  POP_TOP
92  LOAD_CONST        12 ('Good!')
...
```

```
".join(map)
inputSerial
```



## Interpretando bytecode

```
34  LOAD_FAST          0 (inputSerial)
37  LOAD_CONST         2 ('')
40  LOAD_ATTR          2 (join)
43  LOAD_GLOBAL        3 (map)
46  LOAD_CONST         3 (<code object <lambda>>)
49  MAKE_FUNCTION      0
52  LOAD_CONST         4 (225)
55  LOAD_CONST         5 (245)
58  LOAD_CONST         6 (241)
61  LOAD_CONST         7 (230)
64  LOAD_CONST         8 (215)
67  LOAD_CONST         9 (161)
70  LOAD_CONST        10 (202)
73  LOAD_CONST        11 (200)
76  BUILD_LIST         8
79  CALL_FUNCTION       2
82  CALL_FUNCTION       1
-->85  COMPARE_OP         2 (==)
88  JUMP_IF_FALSE      10 (to 101)
91  POP_TOP
92  LOAD_CONST        12 ('Good!')
...
```

".join(map  
inputSerial

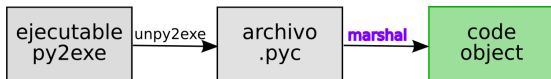


## byteplay

Corrigiendo el bytecode

- **URL:** <http://code.google.com/p/byteplay/>
- **Licencia:** LGPL
- **Autor:** Noam Raph
- **Versión:** 0.2
- Permite convertir code objects en objetos equivalentes pero manipulables, y a su vez volver a convertirlos en code objects.
- También:
  - **BytecodeAssembler**

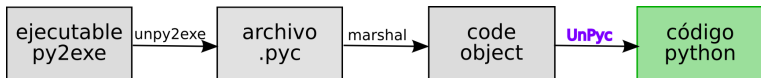
<http://pypi.python.org/pypi/BytecodeAssembler>



# uncompyle

Volviendo a las fuentes... de .pyc a .py

- **URL:** <https://github.com/matiasb/uncompyle> (fork)
- **Licencia:** BSD
- **Autor:** Dmitri Kornev/Günther Starnberger
- **Versión:** 0.11
- Herramienta que permite decompilar archivos .pyc.  
Soporta Python 2.5, 2.6, 2.7



# Python Bytecode

- 1 De .py a .pyc
- 2 Análisis pyc to gráfico
- 3 Experimentos
- 4 Buscando protección



## Por qué?

- Aplicaciones comerciales/cerradas en Python sin permitir acceso al código fuente.
- Existen diferentes técnicas, se pueden combinar.
- En general tienden a ocultar o cambiar el bytecode en disco.





# Alternativas

Usar un packager

- Se llevan los archivos Python a un formato binario nativo.
- Facilita distribución, no Python required.
- Se distribuyen runtime y deps con el código.
- py2exe, py2app, cxFreeze, pyInstaller, entre otros.



# Alternativas

Ofuscar el código fuente

- Se busca ocultar la lógica detrás del código.
- Usado mucho en js.
- No se ve mucho en Python.



## Alternativas

Usar un runtime modificado

### Cambiar magic number

- Se chequea en `import.c`.
- Hace fallar herramientas usuales.
- Pero... bastaría cambiar el magic number a uno válido.

### Cambiar marshalling

- Se modifica `marshal.c`.
- Adicionalmente se puede encriptar el code object serializado.

### Remapear opcodes

- Se modifica `opcodes.h`.
- No se distribuye `opcodes.py` para evitar su uso.



# Preguntas?

**Matías Bordese**

mbordese at gmail dot com

matiasb at freenode

<http://matias.bordese.com.ar>



Licencia: Creative Commons

Atribución-NoComercial-CompartirDerivadasIgual 2.5 Argentina

[http://creativecommons.org/licenses/by-nc-sa/2.5/deed.es\\_AR](http://creativecommons.org/licenses/by-nc-sa/2.5/deed.es_AR)

<https://github.com/matiasb/python-bytecode>



## Ejemplos (no) interactivos

Hello world con BytecodeAssembler

```
>>> from peak.util.assembler import Code
>>>
>>> c = Code()
>>> c.LOAD_CONST('hello world!')
>>> c.PRINT_ITEM()
>>> c.PRINT_NEWLINE()
>>> c.LOAD_CONST(None)
>>> c.RETURN_VALUE()
>>>
>>> exec c.code()
hello world!
```

- Un ejemplo muy básico.
- Recomendable leer la documentación del proyecto.



## Ejemplos (no) interactivos

Retocando el bytecode con byteplay

```
>>> import marshal
>>> f = open('crkm0x1.pyc', 'rb')
>>> magic = f.read(4)
>>> timestamp = f.read(4)
>>> code = marshal.load(f)

>>> import byteplay
>>> bp = byteplay.Code.from_code(code)
>>> main = bp.code[1][1]
>>> main.code[40] = (byteplay.PRINT_ITEM, None)
>>> main.code[41] = (byteplay.PRINT_NEWLINE, None)
>>> exec main.to_code()
Serial: 123
qeavG1ZX
Good!
```

- Tomamos el code object main que extraemos después de obtener el .pyc del crkm0x1.exe con unpy2exe.
- Los offsets modificados (40, 41) corresponden a los opcodes que veíamos aquí.

