

## **Comprehensions: ler, entender e utilizar**

---

Alexandre Yukio Harano

Curitiba, 12 de Setembro de 2019 – Python Sul

`alexandre@harano.net.br`

`https://alexandre.harano.net.br/`

**O que é *comprehension*?**

Uma maneira compacta de processar todos os ou parte dos elementos de uma sequência e devolver (uma lista | um dicionário | um conjunto) com os resultados.

Adaptado de <https://docs.python.org/3/glossary.html#term-list-comprehension>

# Comprehension

```
comprehension ::= expression comp_for
comp_for      ::= ["async"] "for" target_list "in"
               or_test [comp_iter]
comp_iter     ::= comp_for | comp_if
comp_if       ::= "if" expression_nocond [comp_iter]
```

Extraído de <https://docs.python.org/3/reference/expressions.html#list-displays>

## ( List | Dict | Set ) Comprehensions

Expressivo mecanismo para criação de uma nova instância de lista/dicionário/conjunto.

## ( List | Dict | Set ) Comprehensions

Expressivo mecanismo para criação de uma nova instância de lista/dicionário/conjunto..

## ( List | Dict | Set ) Comprehensions

Expressivo mecanismo para criação de uma nova instância de lista/dicionário/conjunto...

## ( List | Dict | Set ) Comprehensions

Expressivo mecanismo para criação de uma nova instância de lista/dicionário/conjunto... se usado cautelosamente.



# ( List | Dict | Set ) Comprehensions

```
lista = [  
    EXPRESSÃO_DO_CONTEÚDO  
    "for" ... "in" ...  
    "if" ...  
]  
  
dicionário = {  
    EXPRESSÃO_DA_CHAVE: EXPRESSÃO_DO_VALOR  
    "for" ... "in" ...  
    "if" ...  
}  
  
conjunto = {  
    EXPRESSÃO_DO_VALOR  
    "for" ... "in" ...  
    "if" ...  
}
```

## ( List | Dict | Set ) Comprehensions

Aloca os recursos necessários para **todos** os elementos que atendam os critérios.

# Expressão Geradora

```
expressão_geradora = (  
    EXPRESSÃO_DO_CONTEÚDO  
    "for" ... "in" ...  
    "if" ...  
)
```

Aloca os recursos necessários **item a item conforme demanda.**

# Por que Expressão Geradora e não Generator Comprehension?

## Resposta:

*Comprehensions* foram pensados inicialmente para *aparentar* com as estruturas de dados construídas (termo original: *display*) e *geradores* não possuem tal relação.

Mais detalhes em [https://nedbatchelder.com/blog/201605/generator\\_comprehensions.html](https://nedbatchelder.com/blog/201605/generator_comprehensions.html)

## Um Pequeno Desvio...

```
>>> lista = [x for x in range(0)]  
>>> lista  
[]  
>>> bool(lista)  
False
```

```
>>> expressão_geradora = (x for x in range(0))  
>>> expressão_geradora  
<generator object <genexpr> at 0x7f4ee64f9bf8>  
>>> bool(expressão_geradora)  
True
```

## ... E Como Contornar

```
>>> expressão_geradora = (x for x in range(0))  
>>> expressão_geradora  
<generator object <genexpr> at 0x7f4ee64f9bf8>  
>>> bool(expressão_geradora)  
True
```

```
>>> lista_da_expressão_geradora =  
...     list(expressão_geradora)  
>>> lista_da_expressão_geradora  
[]  
>>> bool(lista_da_expressão_geradora)  
False
```

# Iteradores Com Variáveis Dependentes

Iterações são lidas da expressão mais a esquerda para a direita.

Exemplo:

```
>>> matriz_triangular_3x3 = [  
...     (x, y)  
...     for y in range(3)  
...     for x in range(y+1)  
... ]  
>>> matriz_triangular_3x3  
[(0, 0), (0, 1), (1, 1),  
 (0, 2), (1, 2), (2, 2)]
```

(0, 0)		
(0, 1)	(1, 1)	
(0, 2)	(1, 2)	(2, 2)



# Asynchronous Comprehensions (PEP 530)

- Só pode ser utilizado dentro de funções com `async def`.
- Expressões podem ser calculadas a partir de
  - i Iteradores assíncronos (`expr async for ...`); ou
  - ii Uso de `await` em *comprehensions* regulares (`await expr for ...`).



# **Análise de código**

dis — Disassembler for Python bytecode — Python 3.7.4 documentation - Firefox Nightly (Private Browsing)

dis — Disassembler for Python bytecode

https://docs.python.org/3/library/dis.html

Python » English 3.7.4 Documentation » The Python Standard Library » Python Language Services » Quick search Go | previous | next | modules | index

Table of Contents

- dis — Disassembler for Python bytecode
  - Bytecode analysis
  - Analysis functions
  - Python Bytecode Instructions
  - Opcodes collections

Previous topic  
compileall — Byte-compile Python libraries

Next topic  
pickletools — Tools for pickle developers

This Page  
Report a Bug  
Show Source

## dis — Disassembler for Python bytecode¶

Source code: [Lib/dis.py](#)

The `dis` module supports the analysis of CPython [bytecode](#) by disassembling it. The CPython bytecode which this module takes as an input is defined in the file `Include/opcode.h` and used by the compiler and the interpreter.

**CPython implementation detail:** Bytecode is an implementation detail of the CPython interpreter. No guarantees are made that bytecode will not be added, removed, or changed between versions of Python. Use of this module should not be considered to work across Python VMs or Python releases.

*Changed in version 3.6:* Use 2 bytes for each instruction. Previously the number of bytes varied by instruction.

Example: Given the function `myfunc()`:

```
def myfunc(alist):
    return len(alist)
```

the following command can be used to display the disassembly of `myfunc()`:

```
>>> dis.dis(myfunc)
2          0 LOAD_GLOBAL              0 (len)
          2 LOAD_FAST                   0 (alist)
          4 CALL_FUNCTION                 1
```

```
>>> def via_nome(): return list()
>>> def via_símbolo(): return []

>>> import dis
>>> dis.dis(via_nome)
1          0 LOAD_GLOBAL              0 (list)
          2 CALL_FUNCTION              0
          4 RETURN_VALUE

>>> dis.dis(via_símbolo)
1          0 BUILD_LIST                0
          2 RETURN_VALUE
```

# Comparação

```
def usual():
    resultado = []
    for valor in range(100):
        if valor % 2:
            resultado.append(valor)
    return resultado

def list_comp():
    return [
        valor
        for valor in range(100)
        if valor % 2
    ]

def filter_lambda():
    return list(
        filter(lambda valor: valor % 2, range(100))
    )
```

# Comparação (usual)

```
>>> dis.dis(usual)
2          0 BUILD_LIST          0
          2 STORE_FAST          0 (resultado)

3          4 SETUP_LOOP          34 (to 40)
          6 LOAD_GLOBAL          0 (range)
          8 LOAD_CONST          1 (100)
         10 CALL_FUNCTION          1
         12 GET_ITER
>>        14 FOR_ITER          22 (to 38)
         16 STORE_FAST          1 (valor)

4          18 LOAD_FAST          1 (valor)
         20 LOAD_CONST          2 (2)
         22 BINARY_MODULO
         24 POP_JUMP_IF_FALSE    14

5          26 LOAD_FAST          0 (resultado)
         28 LOAD_METHOD          1 (append)
         30 LOAD_FAST          1 (valor)
         32 CALL_METHOD          1
         34 POP_TOP
         36 JUMP_ABSOLUTE        14
>>        38 POP_BLOCK

6        >>        40 LOAD_FAST          0 (resultado)
         42 RETURN_VALUE
```



# Comparação (list\_comp)

```
>>> dis.dis(list_comp)
3          0 LOAD_CONST          1 (<code object <listcomp> at 0x7fddefe4ced0, file "<stdin>", line 3>)
          2 LOAD_CONST          2 ('list_comp.<locals>.<listcomp>')
          4 MAKE_FUNCTION          0

4          6 LOAD_GLOBAL          0 (range)
          8 LOAD_CONST          3 (100)
         10 CALL_FUNCTION          1
         12 GET_ITER
         14 CALL_FUNCTION          1
         16 RETURN_VALUE
```

Disassembly of <code object <listcomp> at 0x7fddefe4ced0, file "<stdin>", line 3>:

```
3          0 BUILD_LIST          0
          2 LOAD_FAST          0 (.0)
      >>    4 FOR_ITER          16 (to 22)

4          6 STORE_FAST          1 (valor)

5          8 LOAD_FAST          1 (valor)
         10 LOAD_CONST          0 (2)
         12 BINARY_MODULO
         14 POP_JUMP_IF_FALSE          4
         16 LOAD_FAST          1 (valor)
         18 LIST_APPEND          2
         20 JUMP_ABSOLUTE          4
      >>    22 RETURN_VALUE
```



# Comparação (filter\_lambda)

```
>>> dis.dis(filter_lambda)
2          0 LOAD_GLOBAL              0 (list)

3          2 LOAD_GLOBAL              1 (filter)
          4 LOAD_CONST                  1 (<code object <lambda> at 0x7efd64690a50, file "<stdin>", line 3>)
          6 LOAD_CONST                  2 ('filter_lambda.<locals>.<lambda>')
          8 MAKE_FUNCTION                0
         10 LOAD_GLOBAL              2 (range)
         12 LOAD_CONST                  3 (100)
         14 CALL_FUNCTION                1
         16 CALL_FUNCTION                2
         18 CALL_FUNCTION                1
         20 RETURN_VALUE
```

Disassembly of <code object <lambda> at 0x7efd64690a50, file "<stdin>", line 3>:

```
3          0 LOAD_FAST                0 (valor)
          2 LOAD_CONST                  1 (2)
          4 BINARY_MODULO
          6 RETURN_VALUE
```





# Asynchronous Comprehensions

```
import asyncio

async def intervalo_progressivo(espera, até):
    for i in range(até):
        yield i
        await asyncio.sleep(espera)

async def async_set():
    return {
        valor
        async for valor in intervalo_progressivo(.001, 100)
        if valor % 2
    }
```

# Asynchronous Comprehensions

```
>>> dis.dis(async_set)
10          0 LOAD_CONST          1 (<code object <setcomp> at 0x7f4b340f0a50, file "a
          2 LOAD_CONST          2 ('async_set.<locals>.<setcomp>')
          4 MAKE_FUNCTION          0

12          6 LOAD_GLOBAL          0 (intervalo_progressivo)
          8 LOAD_CONST          3 (0.001)
         10 LOAD_CONST          4 (100)
         12 CALL_FUNCTION          2
         14 GET_AITER
         16 CALL_FUNCTION          1
         18 GET_AWAITABLE
         20 LOAD_CONST          0 (None)
         22 YIELD_FROM
         24 RETURN_VALUE

...
```



# Asynchronous Comprehensions

...

Disassembly of <code object <setcomp> at 0x7f4b340f0a50, file "async\_set.py", line 10>:

```
10      0 BUILD_SET                0
      2 LOAD_FAST                  0 (.0)
    >>  4 SETUP_EXCEPT            12 (to 18)
      6 GET_ANEXT
      8 LOAD_CONST                0 (None)
     10 YIELD_FROM

     12      12 STORE_FAST             1 (valor)
      14 POP_BLOCK
     16 JUMP_FORWARD              10 (to 28)
    >>  18 DUP_TOP
      20 LOAD_GLOBAL                0 (StopAsyncIteration)
      22 COMPARE_OP               10 (exception match)
      24 POP_JUMP_IF_TRUE         42
     26 END_FINALLY
```

...



# Asynchronous Comprehensions

...

```
13      >> 28 LOAD_FAST          1 (valor)
           30 LOAD_CONST        1 (2)
           32 BINARY_MODULO
           34 POP_JUMP_IF_FALSE    4
           36 LOAD_FAST          1 (valor)
           38 SET_ADD           2
           40 JUMP_ABSOLUTE     4
           >> 42 POP_TOP
           44 POP_TOP
           46 POP_TOP
           48 POP_EXCEPT
           50 POP_TOP
           52 RETURN_VALUE
```

# Não-Exemplos (Aprecie Com Moderação!)

```
resultado = [transformação_complexa(  
    x, algum_argumento=x+1)  
    for x in iterável if predicado(x)]
```

```
resultado = [  
    (x, y)  
    for x in range(10)  
    for y in range(5)  
    if x * y > 10  
]
```

```
return ((x, y, z)  
        for x in range(5)  
        for y in range(5)  
        if x != y  
        for z in range(5)  
        if y != z)
```

Adaptado de <https://github.com/google/styleguide/blob/gh-pages/pyguide.md>



<https://github.com/ayharano/just-python/>

## Pontos principais

- *Comprehensions* são utilizados para **criar** listas|dicionários|conjuntos.
- Pode melhorar a legibilidade de código principalmente para filtrar dados.
- Variáveis alocadas dentro da *comprehensions* só valem dentro do escopo local, ou seja, menos variáveis temporárias.
- Não recomendado usar com expressões que exijam tratamento (ex: `Exceptions`).
- Cautela com os recursos! Se souber de antemão que os dados extrapolam a memória, usar *expressões geradoras*.

# Perguntas?

**Alexandre Yukio Harano**

**[alexandre@harano.net.br](mailto:alexandre@harano.net.br)**

**<https://alexandre.harano.net.br/>**