# Superpoderes de iteración

Python Meetup Marzo 2022

Bruno Constanzo

# Vista de pájaro

- Malos hábitos
- ¿Cómo iterar?
- Built-ins e *itertools*
- Generators y comprehensions
- Iterator Protocol

# Referencias


- Ned Batchelder, “Loop Like a Native”
  PyCon US 2013
- Nina Zhakarenko, “Elegant Solutions for Everyday Python Problems”
  PyCon US 2018
- Raymond Hettinger, “Beyond PEP 8”
  PyCon Montreal 2015

# Malos hábitos

En otros lenguajes nos ocupamos directamente de la iteración

# Malos hábitos

En otros lenguajes nos ocupamos directamente de la iteración

```
datos = [1, 2, 3, 4]

i = 0
while i < len(datos):
    d = datos[i]
    print(d)
    i = i + 1
```

# Malos hábitos

En otros lenguajes nos ocupamos directamente de la iteración

```
datos = [1, 2, 3, 4]

i = 0
while i < len(datos):
    d = datos[i]
    print(d)
    i = i + 1
```

```
datos = [1, 2, 3, 4]

for i in range(len(datos)):
    d = datos[i]
    print(d)
```

# Malos hábitos

En otros lenguajes nos ocupamos directamente de la iteración

```
datos = [1, 2, 3, 4]

i = 0
while i < len(datos):
    d = datos[i]
    print(d)
    i = i + 1
```

```
datos = [1, 2, 3, 4]

for i in range(len(datos)):
    d = datos[i]
    print(d)
```

```
datos = [1, 2, 3, 4]

for d in datos:
    print(d)
```

# Malos hábitos

En otros lenguajes nos ocupamos directamente de la iteración

```
datos = [1, 2, 3, 4]

i = 0
while i < len(datos):
    d = datos[i]
    print(d)
    i = i + 1
```

```
datos = [1, 2, 3, 4]

for i in range(len(datos)):
    d = datos[i]
    print(d)
```

```
datos = open(...)

for d in datos:
    print(d)
```

# ¿Cómo iterar?

# ¿Cómo iterar?

Si vamos a iterar en base a una condición, usamos while

```
while condición:
    hacer_algo()
```

# ¿Cómo iterar?

Si vamos a iterar en base a una condición, usamos while

Si queremos recorrer una secuencia , usamos for

```
while condición:
    hacer_algo()
```

```
for elemento in secuencia:
    hacer_algo(elemento)
```

# ¿Cómo iterar?

El elemento que obtenemos depende de la secuencia

```
for c in string:
    print(c)
```

```
for item in lista:
    print(item)
```

```
for línea in archivo:
    print(línea.strip())
```

```
for clave in diccionario:
    print(clave)
```

# Built-ins e *itertools*

Python cuenta con muchas funciones que procesan iterables

```
>>> min([1, 100, -1, 3])
-1
```

```
>>> max(["1", "100", "-1", "3"])
'3'

>>> max(["1", "100", "-1", "3"], key=int)
'100'
```

```
>>> sum([1, 100, -1, 3])
103
```

```
>>> list(filter(lambda x: x >= 0, [1, 100, -1, 3]))
[1, 100, 3]
```

# Built-ins e *itertools*

Python cuenta con muchas funciones que procesan iterables

```
>>> palabras = ["uno", "dos", "tres"]
>>> list(enumerate(palabras))
[(0, 'uno'), (1, 'dos'), (2, 'tres')]
```

```
>>> numeros = [1, 2, 3, 4, 5]
>>> palabras = ["uno", "dos", "tres"]
>>> list(zip(palabras, numeros))
[('uno', 1), ('dos', 2), ('tres', 3)]

>>> list(zip(*zip(numeros, palabras)))
[('uno', 'dos', 'tres'), (1, 2, 3)]
```

# Built-ins e *itertools*

*itertools* tiene:

- accumulate
- compress
- filterfalse
- product
- tee
- chain
- count
- groupby
- repeat
- zip_longest
- combinations
- cycle
- islice
- starmap
- combinations_with_replacement
- dropwhile
- permutations
- takewhile

# Generators y comprehensions

Generator (descripción rápido y mal):

- Es una función que usa yield

# Generators y comprehensions

Generator (descripción rápido y mal):

- Es una función que usa yield
- yield devuelve un valor a next() y deja pausado el generator

# Generators y comprehensions

Generator (descripción rápido y mal):

- Es una función que usa yield
- yield devuelve un valor a next() y deja pausado el generator
- al llamar next() nuevamente continúa luego del yield

# Generators y comprehensions

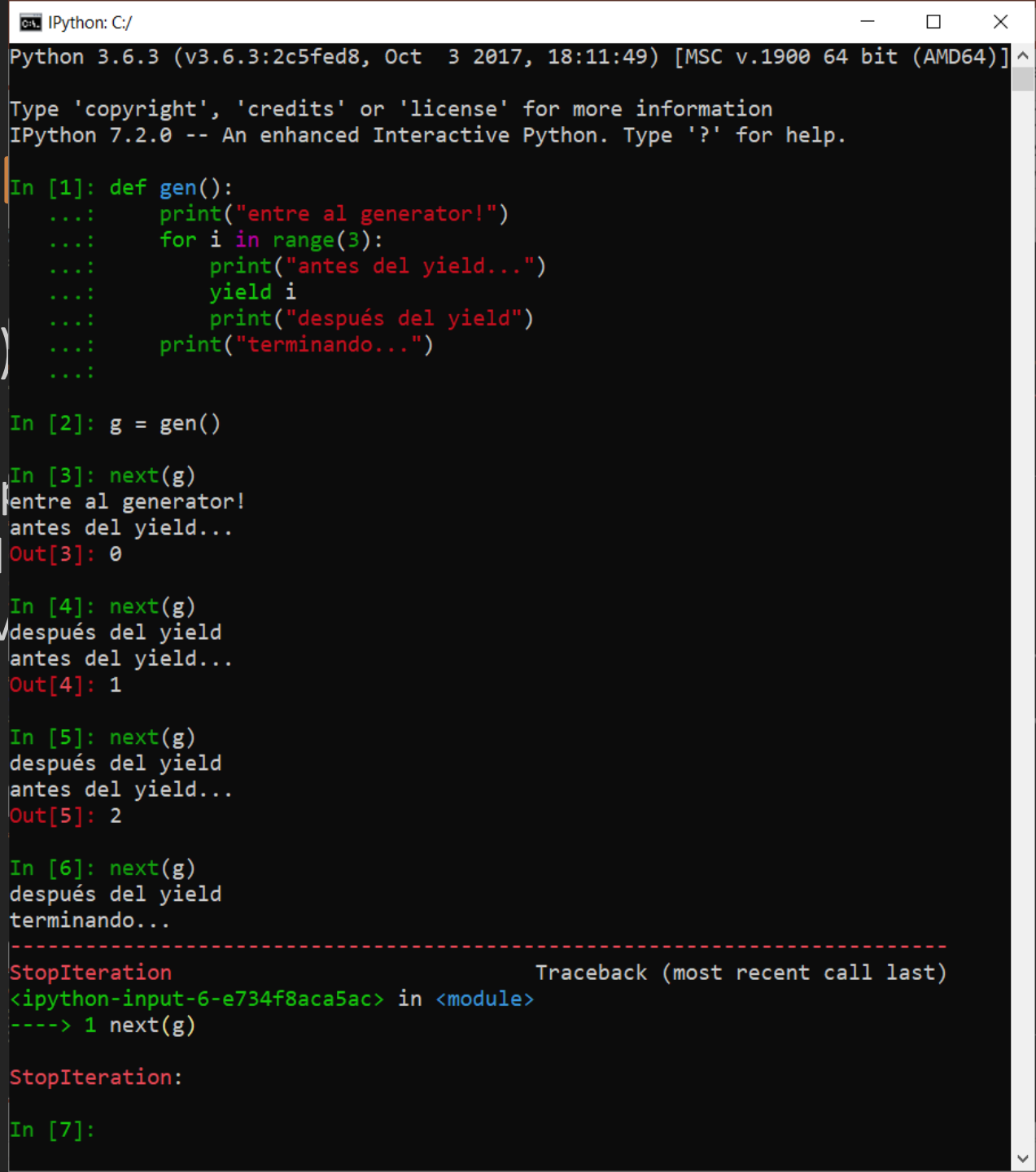
Generator (descripción rápido y mal):

- Es una función que usa yield
- yield devuelve un valor a next() y deja pausado el generator
- al llamar next() nuevamente continúa luego del yield
- cuando nos quedamos sin valores, eleva StopIteration

# Generators y comprehe

Generator (descripción rápido y mal)

- Es una función que usa yield
- yield devuelve un valor a next() y deja
- al llamar next() nuevamente continúa l
- cuando nos quedamos sin valores, elev



# Generators y comprehensions

List comprehension

```
ingredientes = ["asado", "berenjenas", "chernia"]
preparaciones = ["a la parrilla", "al horno"]
comidas = []

for i in ingredientes:
    for p in preparaciones:
        comidas.append(f"{i} {p}")
```

# Generators y comprehensions

List comprehension

```
ingredientes = ["asado", "berenjenas", "chernia"]
preparaciones = ["a la parrilla", "al horno"]
comidas = []

for i in ingredientes:
    for p in preparaciones:
        comidas.append(f"{i} {p}")

comidas = [ f"{i} {p}" for i in ingredientes for p in preparaciones ]
```

# Generators y comprehensions

List comprehension

```
ingredientes = ["asado", "berenjenas", "chernia"]
preparaciones = ["a la parrilla", "al horno"]
comidas = []

for i in ingredientes:
    for p in preparaciones:
        comidas.append(f"{i} {p}")

comidas = [ f"{i} {p}" for i in ingredientes for p in preparaciones ]

comidas = [
    f"{i} {p}"
    for i in ingredientes
    for p in preparaciones
]
```

# Generators y comprehensions

List comprehension -> generator expression

```
ingredientes = ["asado", "berenjenas", "chernia"]
preparaciones = ["a la parrilla", "al horno"]

def generator_comidas():
  for i in ingredientes:
    for p in preparaciones:
        yield f"{i} {p}"

comidas = ( f"{i} {p}" for i in ingredientes for p in preparaciones )

comidas = (  # ahora es una generator expression
    f"{i} {p}"
    for i in ingredientes
    for p in preparaciones
)
```

# Iterator Protocol

# Iterator Protocol

Contenedor:

- `__iter__()` que devuelve un iterator

# Iterator Protocol

Contenedor:

- __iter__() que devuelve un iterator

Iterator:

- __iter__() que devuelve self
- __next__() que devuelve el siguiente valor o eleva StopIteration

# for e in container:

¿Qué pasa cuando hacemos un for en python?

# for e in container:

¿Qué pasa cuando hacemos un for en python?

```
// hagamos de cuenta que es C
for(i=0; i < 10; i++)
{
    e = contenedor[i];
    ... // bloque del for
}
```

# for e in container:

¿Qué pasa cuando hacemos un for en python?

```
// hagamos de cuenta que es C
for(i=0; i < 10; i++)
{
    e = contenedor[i];
    ... // bloque del for
}
```

```
index = 0
last = len(secuencia)
while index < last:
    e = secuencia[i]
    index = index + 1
    # acá vendría el bloque
    print(e)
```

# for e in container:

¿Qué pasa cuando hacemos un for en python?

```
// hagamos de cuenta que es C
for(i=0; i < 10; i++)
{
    e = contenedor[i];
    ... // bloque del for
}
```

```
index = 0
last = len(secuencia)
while index < last:
    e = secuencia[i]
    index = index + 1
    # acá vendría el bloque
    print(e)
```

```
it = iter(contenedor)
while True:
    try:
        e = next(it)
    except StopIteration:
        break
    # acá viene el bloque del for
    print(e)
```

# Curiosidades

```python
params = {
    "query": QUERY,
    "page_size": 100,
}

# Get page=0, page=1, page=2, ...
for params["page"] in itertools.count():
    data = requests.get(SEARCH_URL, **params).json()
    if not data["results"]:
        break
    ...
```

Ned Batchelder
@nedbat

For loops in #python can assign to more than just plain names. Any assignment target is allowed.

Traducir Tweet

11:52 a. m. · 28 feb. 2022 · Twitter Web App

73 Retweets 16 Tweets citados

599 Me gusta

Twittea tu Responder

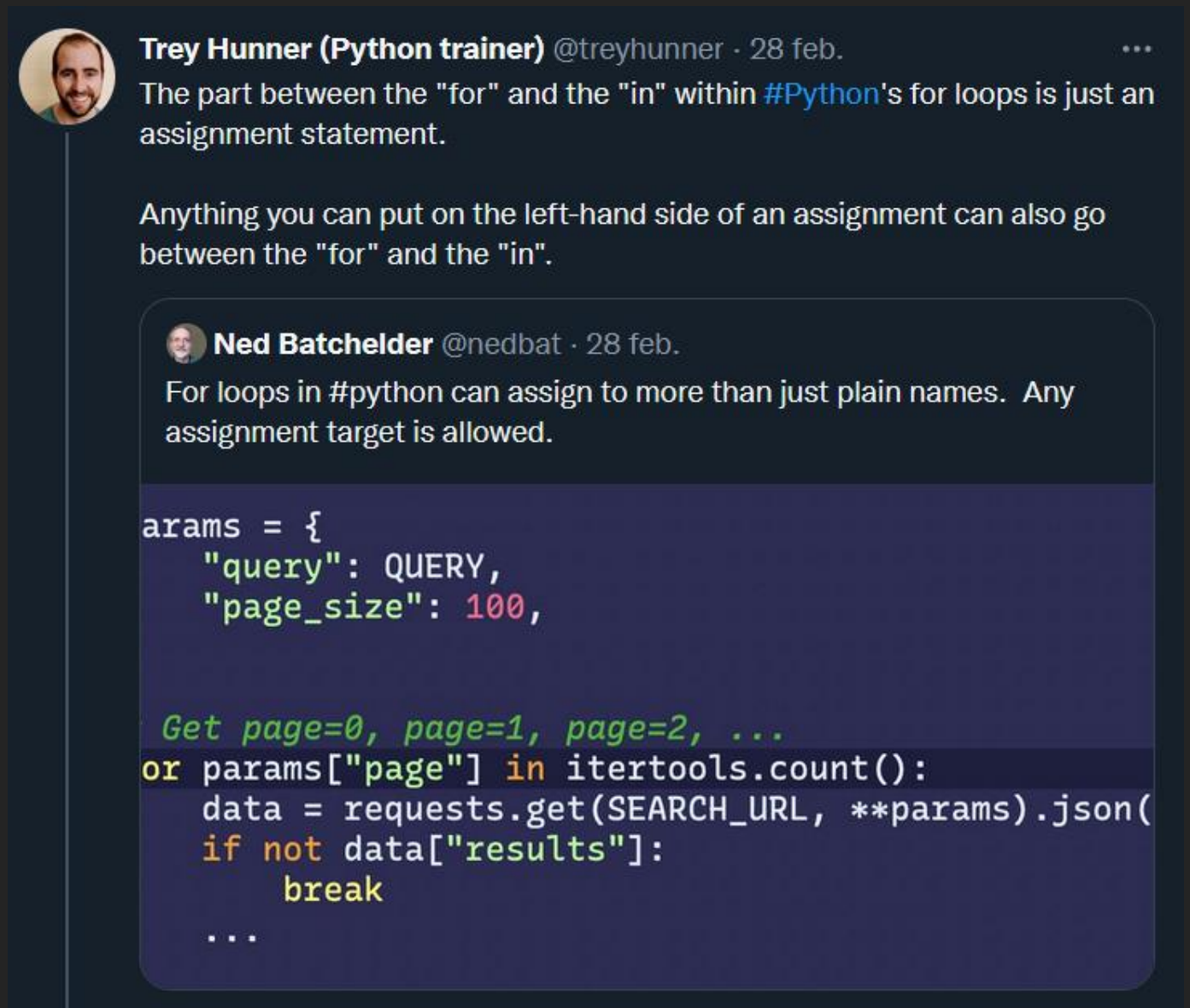
Armin ... @mit... · 28 feb.
En respuesta a @nedbat
fun fact: jinja at one point used to compile down to exactly that to assign to `context['var']`. However it's comparatively very slow, which is why it was changed

25

Seth Mic... @seth... · 28 feb.
En respuesta a @nedbat
Is that **params supposed to be params=params?

1 5

# Curiosidades



**Trey Hunner (Python trainer)** @treyhunner · 28 feb.

The part between the "for" and the "in" within #Python's for loops is just an assignment statement.

Anything you can put on the left-hand side of an assignment can also go between the "for" and the "in".

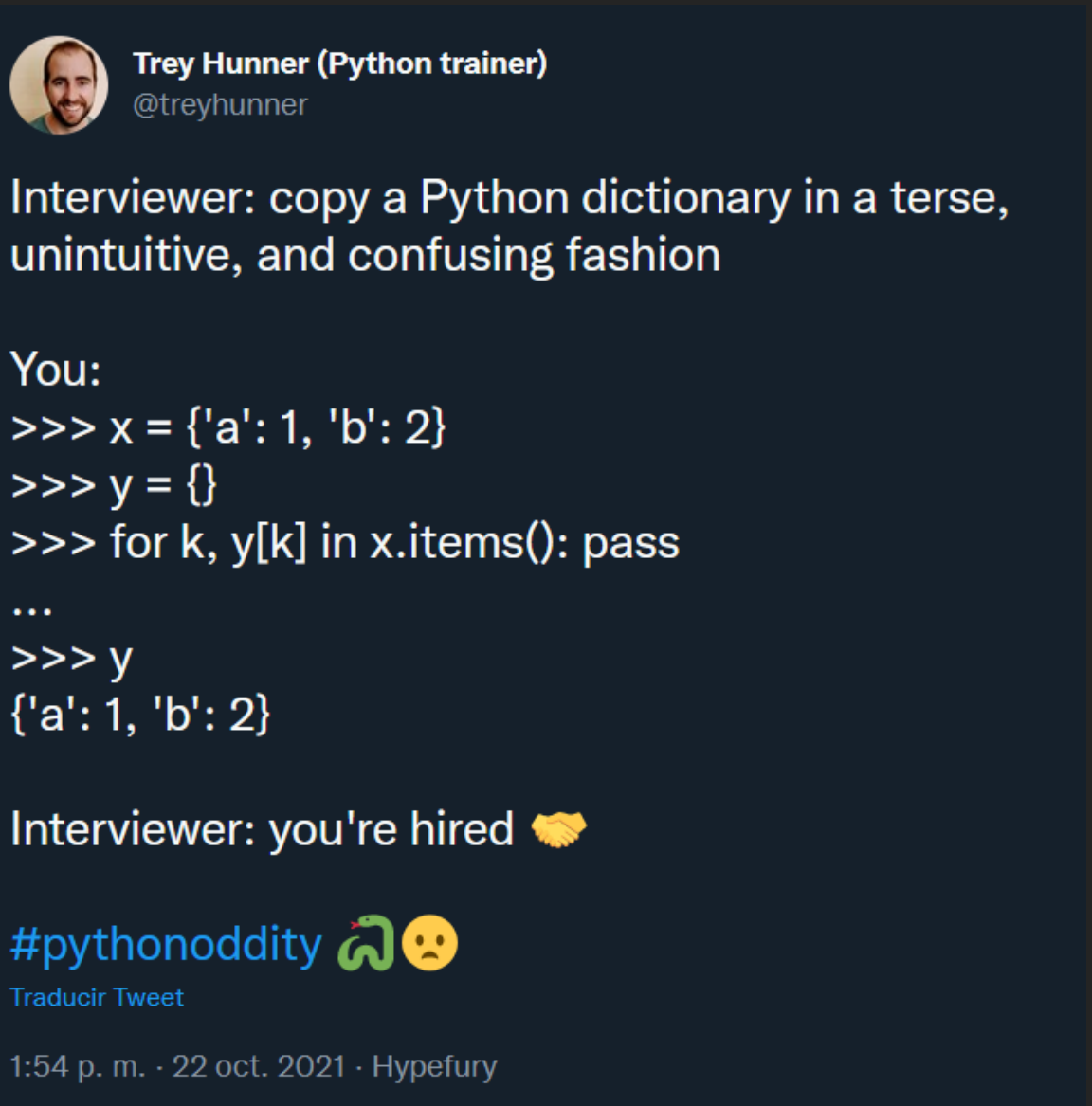
**Ned Batchelder** @nedbat · 28 feb.

For loops in #python can assign to more than just plain names. Any assignment target is allowed.

```
arams = {
    "query": QUERY,
    "page_size": 100,


 Get page=0, page=1, page=2, ...
or params["page"] in itertools.count():
    data = requests.get(SEARCH_URL, **params).json(
    if not data["results"]:
        break
    ...
```

# Curiosidades



**Trey Hunner (Python trainer)**
@treyhunner

Interviewer: copy a Python dictionary in a terse, unintuitive, and confusing fashion

You:

```
>>> x = {'a': 1, 'b': 2}
>>> y = {}
>>> for k, y[k] in x.items(): pass
...
>>> y
{'a': 1, 'b': 2}
```

Interviewer: you're hired 🤝

#pythonoddity 🐍😕

Traducir Tweet

1:54 p. m. · 22 oct. 2021 · Hypefury

# ¿Preguntas?

¿Alguna?

¿Nada?

¡Muchas gracias por su tiempo!

¡Y vean las charlas de Ned, Nina y Raymond!