Assignment expressions do zero ao anti-herói

O novo "walrus" do Python 3.8 – PEP 572

Danilo J. S. Bellini
@danilobellini

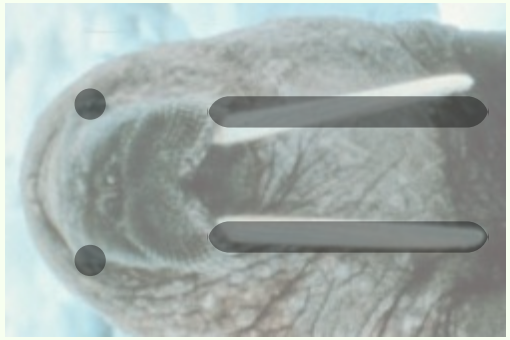
:= :=

2019-11-09

Just Python 3.0 @ Creditas – SP

Walrus – O operador das *assignment expressions* (PEP 572)

Veremos sobre o operador "`:=`", conhecido como *Walrus* (Morsa)



O que faz esse operador?

Esse operador faz com que uma atribuição possa ser realizada em meio a uma expressão, não mais apenas como *statement*.

```
>>> import re
>>> msg = "Just Python 3.0 terá lightnings + 12 palestras!"

# Vamos encontrar um número de pelo menos 2 dígitos em msg
>>> match = re.search(r"\d{2,}", msg)
>>> if match:
...     print(match.group())
12

# A tentação: duplicar código...
>>> if re.search(r"\d{2,}", msg):
...     print(re.search(r"\d{2,}", msg).group())
12

# Com assignment expression
>>> if match := re.search(r"\d{2,}", msg):
...     print(match.group())
12
```

Evitando atribuições desnecessárias

Supondo que temos `p1` e `p2` como *regexes*, por exemplo `re.compile(r"^Just\s?Python\s?\S+")` e `re.compile(r"\S{5,}")`.

Greedy/eager

```
m1 = p1.search(msg)
m2 = p2.search(msg)
if m1:
    r = m1.group()
elif m2:
    r = m2.group()
else:
    r = "fallback"
```

Só o necessário

```
m1 = p1.search(msg)
if m1:
    r = m1.group()
else:
    m2 = p2.search(msg)
    if m2:
        r = m2.group()
    else:
        r = "fallback"
```

Com o *walrus*

```
if m1 := p1.search(msg):
    r = m1.group()
elif m2 := p2.search(msg):
    r = m2.group()
else:
    r = "fallback"
```

Quando queremos fazer várias verificações consecutivas evitando realizar atribuições desnecessariamente, tínhamos que alterar a estrutura do código ou usar algum tipo de quebra, como um `return`, o que também poderia significar uma mudança na estrutura do código. As *assignment expressions* permitem o mesmo resultado sem gambiarras.

Evitando excesso de aninhamentos / quebra da "linearidade"

```
reductor = dispatch_table.get(cls)
if reductor:
    rv = reductor(x)
else:
    reductor = getattr(x, "__reduce_ex__", None)
    if reductor:
        rv = reductor(4)
    else:
        reductor = getattr(x, "__reduce__", None)
        if reductor:
            rv = reductor()
        else:
            raise Error("un(shallow)copyable object")

# O mesmo (baseado na stdlib "copy"), com o walrus
if reductor := dispatch_table.get(cls):
    rv = reductor(x)
elif reductor := getattr(x, "__reduce_ex__", None):
    rv = reductor(4)
elif reductor := getattr(x, "__reduce__", None):
    rv = reductor()
else:
    raise Error("un(shallow)copyable object")
```

Exemplos de casos de uso

```python
# Utilização de um match de uma expressão regular
if (match := re.search(pattern, data)) is not None:
    # Fazer algo com o match

# Laço que não tem como ser trivialmente escrito de uma vez
while chunk := file.read(8192):
    process(chunk)

# Reuso de um valor caro de ser computado
# (e.g. dentro de um lambda ou de outra expressão)
[y := f(x), y**2, y**3]

# Compartilhamento de subexpressão do filtro da comprehension
# (isto será melhor detalhado adiante)
filtered_data = [y for x in data if (y := f(x)) is not None]
```

```python
# Segundo exemplo sem walrus
chunk = file.read(8192):
while chunk:
    process(chunk)
    chunk = file.read(8192)
```

```python
while True:
    chunk = file.read(8192):
    if not chunk:
        break
    process(chunk)
```

Recomendações de "estilo"

A PEP 572 traz duas "recomendações de estilo", situações em que o uso de *statements* é indicado ao invés de *assignment expressions*:

- Quando "tanto faz"
- Quando houver ambiguidade relativa à ordem de execução

No apêndice A da PEP 572, há diversos exemplos de código cuja legibilidade foi considerada "melhor" ou "pior" por Tim Peters. Exemplos de situações distintas:

```
# Confuso! O "total" aparece muitas vezes em um lugar só
while total != (total := total + term):
    term *= mx2 / (i*(i+1))
    i += 2
return total

# Ok! Mais legível que aninhamentos artificiais em statements
if (diff := x - x_base) and (g := gcd(diff, n)) > 1:
    return g
```

Exemplo final do apêndice A da PEP 572 (adaptado)

Este é um algoritmo para calcular $\lfloor \sqrt[n]{x} \rfloor$, em que os valores de x e n são conhecidos (e inteiros positivos), iniciando com um palpite maior ou igual ao resultado, usando apenas inteiros positivos.

```
>>> x, n, a = 28, 3, 1000  # Com walrus
>>> while a > (d := x // a ** (n - 1)):  # while too big:
...     a = ((n - 1) * a + d) // n       #     a = new guess
>>> a
3

>>> x, n, a = 28, 3, 1000  # Sem walrus
>>> while True:
...     d = x // a ** (n - 1)
...     if a <= d:
...         break
...     a = ((n - 1) * a + d) // n
>>> a
3
```

Qual implementação é mais legível?

[*Slide extra*] Prova do algoritmo do *slide* anterior

Queremos calcular $\lfloor \sqrt[n]{x} \rfloor$, a partir de um valor $a_k \in \mathbb{N}^*$, $a_k \geq \sqrt[n]{x}$. Isso significa que $a_k^n \geq x$. Suponha que o fator que garante a igualdade é d_k/a_k:

$$a_k^n \frac{d_k}{a_k} = x \quad \Rightarrow \quad a_k^{n-1} d_k = x \quad \Rightarrow \quad d_k = \frac{x}{a_k^{n-1}}$$

Idealmente, a média geométrica é Como temos o valor de x, podemos calcular um d_k

Uso em *list/set/dict comprehensions* e *generator expressions*

O *walrus* pode aparecer em *quase* qualquer lugar em que uma expressão é permitida, embora ele tenha de estar com parênteses em algumas situações. É importante saber a ordem de execução!

```
>>> data = ["  a", "lgumas ", "  ", " strings  ", " ", "aqui"]

>>> # Ruim! O ".strip()" é calculado duas vezes
>>> [value.strip() for value in data if value.strip()]
['a', 'lgumas', 'strings', 'aqui']

>>> # Muito melhor!
>>> [trimmed for value in data if (trimmed := value.strip())]
['a', 'lgumas', 'strings', 'aqui']

>>> # Em dicionários, a chave é avaliada antes do valor
>>> {(t := value.strip()): len(t) for value in data}
{'a': 1, 'lgumas': 6, '': 0, 'strings': 7, 'aqui': 4}
```

No iterável (a parte imediatamente após o `in`), é proibido o uso de *assignment expressions* (`SyntaxError`: *assignment expression cannot be used in a comprehension iterable expression*).

Scan (função de ordem superior) / *accumulate*

Esse operador faz com que seja possível implementar uma *comprehension* que acessa o valor da iteração anterior (um "acumulador"), possibilitando que o *scan* da programação funcional seja implementado por meio de uma *list comprehension*.

```
>>> # Soma acumulada
>>> acc = 0
>>> [acc := acc + v for v in [1, 2, 3, 4, -2, 1, 1]]
[1, 3, 6, 10, 8, 9, 10]
>>> v  # O escopo é somente o interno à list comprehension
Traceback (most recent call last):
  ...
NameError: name 'v' is not defined

>>> acc  # Mas, para o walrus, o menor escopo possível
...      # é o externo à list comprehension
10

>>> # Fibonacci (scan e map em uma só list comprehension)
>>> mem = 0, 1
>>> [(mem := (mem[1], sum(mem)))[0] for unused in range(14)]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
```

Escopo e casos não inicializados!

```
# O nome precisa ser definido de antemão para ser utilizável
>>> [undef := None if undef else True for unused in range(9)]
Traceback (most recent call last):
  ...
NameError: name 'undef' is not defined

>>> # O escopo padrão é local (o menor)
>>> unused = [undef := x for x in range(3) if (tr00 := True)]
>>> undef, tr00
(2, True)

>>> # Mas é mantido o escopo original do nome, se for outro
>>> def some_func():
...     global new_name  # O mesmo vale para nonlocal
...     [new_name := x for x in range(19) if (other := True)]
>>> some_func()
>>> new_name
18
>>> other
Traceback (most recent call last):
  ...
NameError: name 'other' is not defined
```

Escopo e `lambda`

Em todos os casos, inclusive do `lambda`, o *walrus* honra o escopo, e não há nenhuma mudança nas regras de escopo do Python por conta do novo operador.

```
# O lambda define um escopo próprio!
>>> p = r"\d+(.*)"
>>> (lambda d: (m := re.match(p, d)) and m.group(1)
... )("123abc")
'abc'
>>> m
Traceback (most recent call last):
  ...
NameError: name 'm' is not defined
```

A **única** exceção é o caso já visto das *list/set/dict comprehensions* e *generator expressions*, as quais o escopo mínimo é o imediatamente externo a elas.

Sintaxe – Onde posso usar?

A atribuição só pode ser feita para um **nome**, e a *assignment expression* também pode ser chamada de *named expression*.

```
# Cuidado! Para o :=, as vírgulas separam as atribuições!
>>> a, b = 7, 0   # Imensa diferença entre o = e o :=
>>> [(a, b := b + a, b - a) for unused in range(4)]
[(7, 7, 0), (7, 14, 7), (7, 21, 14), (7, 28, 21)]

>>> [((a, b) := (b + a, b - a)) for unused in range(4)]
Traceback (most recent call last):
  ...
SyntaxError: cannot use named assignment with tuple

# Não pode o nome usado no "for" de uma list comprehension
>>> [i := i for i in range(10)]
Traceback (most recent call last):
  ...
SyntaxError: assignment expression cannot rebind                    ...
↪ comprehension iteration variable 'i'
```

Em muitos casos, é necessário o uso de parênteses, por exemplo quando se quer atribuir uma `tuple` a um dado nome.

Além das *tuplas*, quando usar parênteses?

```
# Quando for uma statement
y := f(x)    # INVÁLIDO

# Quando estiver no mesmo "nível" de outra atribuição
y0 = y1 := f(x)              # INVÁLIDO
foo(x = y := f(x))           # INVÁLIDO
def foo(answer = p := 42):   # INVÁLIDO
y = (a := b := c)            # INVÁLIDO

# No mesmo nível de um ":", como em annotations e lambdas
{x := y + 2 : y for y in range(3)}  # INVÁLIDO
(lambda: x := 1)                    # INVÁLIDO
def foo(answer: p := 42 = 5):       # INVÁLIDO
def foo(answer: (p := 42) = 5):     # Válido
```

Essa lista não é exaustiva, há outros lugares em que é necessário colocar parênteses, como no `if` de uma *list comprehension* e dentro de *f-strings*. Em todos esses exemplos, colocar parênteses torna a sintaxe válida.

Desafios

```python
# O que isto faz?
if any(len(longline := line) >= 100 for line in lines):
    print("Extremely long line:", longline)

# Qual a diferença entre essas duas listas?
# Se f = lambda k: max(k ** 2, 1), quais são os resultados?
[(lambda y: [y, x / y])(f(x)) for x in range(5)]
[[y := f(x), x / y] for x in range(6)]

# Qual o valor de x em cada um desses casos?
x = (x := 2) + 1
x = (x := ["just"]) + (x := ["python"])
z = (x := ["just"]) + (x := ["python"])
```

Cadê o anti-herói?

O anti-herói *não* é esse novo operador *walrus* e nem mesmo algum potencial uso dele de maneira despreocupada com a legibilidade. O real anti-herói é a **negatividade** de certas interações sociais!

No dia 2018-07-12, GvR enviou uma mensagem na lista `python-committers` c/ o título *Transfer of power*, começando com:

> "*Now that PEP 572 is done, I don't ever want to have to fight so hard for a PEP and find that so many people despise my decisions..*
> *I would like to remove myself entirely from the decision process. I'll still be there for a while as an ordinary core dev, and I'll still be available to mentor people – possibly more available. But I'm basically giving myself a permanent vacation from being BDFL, and you all will be on your own.*"

História (Parte 1/2)

- (2007-05-07) *Issue* 1714448 propõe `if expr as name:`
- (2009-03-14) Discussão é iniciada na `python-ideas`, criação da PEP 379
- (2009-03-15) "*a solid -1 from me*" (GvR),
- (2018-02-27) Definição da PEP 572 por Chris Angelico, com o título *Syntax for Statement-Local Name Bindings*, e início de novas discussões na `python-ideas` a partir de uma prova de conceito que ele próprio criou
- (2018-02-28) Data oficial da efetiva criação da PEP 572
- (2018-03-23) Christoph Groth propõe o uso do `:=`como operador; e "*I also think it's fair to at least reconsider adding inline assignment, with the "traditional"semantics (possibly with mandatory parentheses). This would be easier to learn and understand for people who are familiar with it from other languages (C++, Java, JavaScript).*" (GvR)

História (Parte 1/2)

- (2018-04-11) O título da PEP 572 passa a ser *Assignment Expressions*
- (2018-04-15) "*I strongly prefer* := *as inline assignment operator* (GvR)
- (2018-05-08) GvR define as regras de escopo relativas ao uso do novo operador em *comprehensions*
- (2018-07-11) GvR aprova a PEP 572
- (2018-07-12) GvR abandona o cargo de BDFL

FIM!

https://github.com/
danilobellini/slides-latex

Para rodar os testes dos slides:

```
python3.8 -m doctest slides.tex
```

Referências:

- https://bugs.python.org/issue1714448
- https://www.python.org/dev/peps/pep-0379/
- https://www.python.org/dev/peps/pep-0572/
- https://github.com/python/peps/commits/master?path[]=pep-0572.rst
- https://mail.python.org/pipermail/python-ideas/2018-February/049041.html
- https://mail.python.org/pipermail/python-ideas/2009-March/003423.html
- https://mail.python.org/pipermail/python-ideas/2009-March/003439.html
- https://mail.python.org/pipermail/python-ideas/2018-March/049408.html
- https://mail.python.org/pipermail/python-ideas/2018-March/049409.html
- https://mail.python.org/pipermail/python-ideas/2018-April/049758.html
- https://mail.python.org/pipermail/python-ideas/2018-April/049957.html
- https://mail.python.org/pipermail/python-ideas/2018-May/050456.html
- https://mail.python.org/pipermail/python-dev/2018-July/154601.html
- https://mail.python.org/pipermail/python-committers/2018-July/005664.html