

Python Tips #1

Wim R.M. Cardoen

Email: \$(prefix)[at]gmail[dot]com

where

prefix='wcardoen'

August 25, 2020

In the following paragraphs we discuss some Python tricks. All code snippets below require Python ≥ 3.8 . They can be downloaded from the following location:

<https://github.com/wcardoen/python-reflections>.

- F-strings:

The `format` method for the string class has been around since Python 2.7. This is a familiar pattern for languages of which the syntax has been modelled on the C language.

```
# Example 1: format method since Python 2.7
artist={"von Beethoven":"Bonn",
        "de Balzac":"Tours",
        "van Eyck":"Maaseik",
        "Alighieri":"Firenze"}
for person in artist.keys():
    print("{0:>13} was born in {1}".format(person, artist[person]))
```

The aforementioned lines results in the following output:

```
von Beethoven was born in Bonn
de Balzac was born in Tours
van Eyck was born in Maaseik
Alighieri was born in Firenze
```

In Python 3.6 the formatted string literal (**f-string**) was introduced. In an f-string the following modifications have been applied to a regular string (and its `format` method):

1. the `format` method applied to the regular string is omitted.
2. the regular string must be prefixed by the 'f' or 'F' character.
3. the indices referring to variables in the format method are replaced by the variables as such.

Therefore, the introduction of the f-string makes codes shorter and more readable.

```
# Example 1: use of f-strings (iter 1)
for person in artist.keys():
    print(f"{person:>13} was born in {artist[person]}")
```

The format specifiers can also contain evaluated expressions:

```
# Example 1: use of f-strings (iter 2)
WIDTH = max((len(item) for item in artist.keys()))
for person in artist.keys():
    print(f"{person:>{WIDTH}} was born in {artist[person]}")
```

Since [Python 3.8](#) f-strings support the '=' character for self-documenting expressions and debugging. The f-string `f"{expr=}"` will print the string 'expr=' and suffix it with the evaluated value of the expression 'expr'.

```
# Example 2: Self-documenting expression (iter 1)
from math import cos, pi
print(f" {cos(pi/4.0)=}")
```

This results into the following output:

```
cos(pi/4.0)=0.7071067811865476
```

Applying a format specifier to the previous example:

```
# Example 2: Self-documenting expression using a format specifier (iter 2)
WIDTH=10
PRECISION=4
from math import cos, pi
print(f" {cos(pi/4.0)={WIDTH}.{PRECISION}}")
```

We now get:

```
cos(pi/4.0)=    0.7071
```

For more info: [PEP-0498](#).

- Chaining of comparison operators:

Let $X := \{x_1, x_2, x_3, \dots, x_n\}$ be the set of Python expressions and $P := \{\widehat{op}_1, \widehat{op}_2, \dots, \widehat{op}_{n-1}\}$ be the set of Python comparison operators¹ applied to X .

The compound logical expression:

$$x_1 \widehat{op}_1 x_2 \text{ and } x_2 \widehat{op}_2 x_3 \text{ and } \dots \text{ and } x_{(n-1)} \widehat{op}_{(n-1)} x_n$$

and

$$x_1 \widehat{op}_1 x_2 \widehat{op}_2 x_3 \widehat{op}_3 \dots \widehat{op}_{(n-1)} x_n \quad (1)$$

are equivalent in the Python language.

Example 1::

```
a,b,c=5,3,7
print(f"{a<b<c=}")
# Equivalent to: (a<b) and (b<c)
#                FALSE and TRUE => FALSE
```

which returns:

```
a<b<c=False
```

Example 2::

```
print(f"{15%4==3>2=}")
# Equivalent to: (15%4==3) and (3>2)
#                or:   TRUE   and TRUE => TRUE
```

¹The Python language has the following comparison operators:<, >, ==, >=, <=, !=, is (not), (not)in

which returns:

```
15%4==3>2=True
```

Example 3::

```
lstA=[[1,2],["hello","world"]]
lstB=lstA
lstB[1][0]="HELLO"
lstC=lstB[:]
print(f"{lstA is lstB is lstC=}")
print(f"{lstA == lstB == lstC=}")
# Equivalent to: (lstA is lstB) and (lstB is lstC)
#               or: TRUE and FALSE => FALSE
# Equivalent to: (lstA == lstB) and (lstB == lstC)
#               or: TRUE and TRUE => TRUE
```

which returns:

```
lstA is lstB is lstC=False
lstA == lstB == lstC=True
```

- The `itertools` module:

The `enumerate` method generates the indices and values of an iterable's elements.

```
a= ['a','b','c']
b= ['x','y','z']
c= [10,100]
# enumerate: returns indices and el.
for index, item in enumerate(b):
    print(f"  El. with {index=} is {item=}")
```

```
El. with index=0 is item='x'
El. with index=1 is item='y'
El. with index=2 is item='z'
```

Python's `zip` function generates an iterable of tuples which is derived from its constituent iterables. The newly created iterable ends as soon as one of its constituent iterables is exhausted.

```
for item in zip(a,b,c):
    print(f"  {item=}")
```

```
item=('a', 'x', 10)
item=('b', 'y', 100)
```

The `zip` method allows the extension of `enumerate` to several iterables:

```
for item in zip(range(len(b)),b,c):
    print(f"  {item=}")
```

```
item=(0, 'x', 10)
item=(1, 'y', 100)
```

Python's `itertools` module covers a wide range of functionality. The following 2 examples use functions from this [module](#).

The function `zip_longest` generates (like `zip`) an iterable of tuples derived from its constituent iterables. The new created iterable is exhausted when its **longest** constituent iterable is exhausted. The "empty" elements are filled up with `None`.

```
import itertools as it
for item in it.zip_longest(a,b,c):
    print(f"    {item=}")
```

```
item=('a', 'x', 10)
item=('b', 'y', 100)
item=('c', 'z', None)
```

The `product` function generates a Cartesian product iterable based on its constituent iterables.

```
s = it.product(a,b,c)
for item in s:
    print(f"    {item=}")
```

```
item=('a', 'x', 10)
item=('a', 'x', 100)
item=('a', 'y', 10)
item=('a', 'y', 100)
item=('a', 'z', 10)
item=('a', 'z', 100)
item=('b', 'x', 10)
item=('b', 'x', 100)
item=('b', 'y', 10)
item=('b', 'y', 100)
item=('b', 'z', 10)
item=('b', 'z', 100)
item=('c', 'x', 10)
item=('c', 'x', 100)
item=('c', 'y', 10)
item=('c', 'y', 100)
item=('c', 'z', 10)
item=('c', 'z', 100)
```

Note that the result of the last block code can also be generated using the following tuple comprehension:

```
s= ((x,y,z) for x in a for y in b for z in c)
for item in s:
    print(f"    {item=}")
```