Advanced Python drops

Rodrigo Girão Serrão

Contents

5 – First element that satisfies a condition

When you have an iterable and need to find the first element that satisfies a condition, you can use a generator expression and the built-in `next` to fetch that first element.

The generic recipe looks like this:

```
first = next(elem for elem in iterable if condition(elem))
```

This is a good idea because the generator expression/`next` combo ensures you only search until you find the element you care about. This means that you don't have to compute the condition on the values that come after the value that you wanted.

```
important_numbers = [42, 73, 10, 16, 0]
print(
    next(n for n in important_numbers if n % 2)
)  # 73
```

If there's no such element, you'll either

1. get a `StopIteration` you need to handle:

```
important_numbers = [42, 10, 16, 0]
try:
    print(
        next(n for n in important_numbers if n % 2)
    )
except StopIteration:
    print("No odd numbers found!")
# No odd numbers found!
```

2. pass a default/sentinel value to `next` as its second argument:

```
important_numbers = [42, 10, 16, 0]
print(
    next(
        n for n in important_numbers if n % 2,
        None,
    )
)  # None
```

17 – Create context managers with `contextlib.contextmanager`

The module `contextlib` provides a decorator `contextmanager` that you can use to implement your own context managers.

For that, you just create a generator that yields once. The code before the `yield` is the setup and the code after the `yield` is the cleanup.

Whatever you yield (if anything) can be captured by the `as ...` part of the `with` statement.

Here is an example that reimplements the built-in `open`:

```
from contextlib import contextmanager

@contextmanager
def my_open(path, mode):
    try:
        file = open(path, mode)
        yield file
    finally:
        file.close()
```

The trick is that the `finally` will ensure we close the file, regardless of whether there is an error while working with the open file.

26 – Notify parent class when subclassing

The dunder class method `__init_subclass__` can be used to notify a class when it's subclassed. This is effective for some metaprogramming without having to resort to metaclasses.

In this example, the class `ParentCls` will print a message whenever it is subclassed:

```
class ParentCls:
    def __init_subclass__(cls, **kwargs):
        print(f"{cls} created with {kwargs = }")
```

The argument `cls` will be the subclass, and the keyword arguments `kwargs` come from the subclass definition:

```
class ChildCls(ParentCls, example=True):
    pass
```

When the class `ChildCls` is created, the parent class automatically prints the following:

```
<class '__main__.ChildCls'> created with kwargs = {'example': True}
```

41 – Longest and shortest

The built-ins `max` and `min` have a keyword argument `key` that lets you change the frame of reference for comparisons.

If you then use `functools.partial` to attach another function to `key`, you essentially build new functions just like LEGOs.

My two favourite examples:

1. `max` + `key=len` builds the function `longest`; and
2. `min` + `key=len` builds the function `shortest`.

```
from functools import partial

longest = partial(max, key=len)
shortest = partial(min, key=len)

words = "This is a truly extraordinary sentence".split()
print(longest(words))  # extraordinary
print(shortest(words))  # a
```

46 – Dot product idiom

The dot product is a mathematical operation that can be computed with a simple Python idiom using the module `operator`:

```
import operator

sum(map(operator.mul, vec1, vec2))
```

The snippet above assumes `vec1` and `vec2` are iterables that represent vectors. This idiom was present in the documentation of the module `itertools` up to Python 3.11.

Then, in Python 3.12, the built-in `zip` got the keyword argument `strict`, which made the idiom evolve into something that looks a bit more complicated:

```
from itertools import starmap
import operator

sum(starmap(operator.mul, zip(vec1, vec2, strict=True)))
```

We're using `zip(..., strict=True)` to ensure the vectors have the same size and `zip` produces tuples, so `starmap` is being used to "unpack" that tuple into the two arguments to `operator.mul`.

Then, in Python 3.14, the built-in `map` got a similar keyword argument `strict`, which means the idiom can go back to its simpler form with the extra safety check:

```
import operator

sum(map(operator.mul, vec1, vec2, strict=True))
```