Intermediate Python drops

Rodrigo Girão Serrão

Contents

8 – Schedule cleanup actions

If you need to clean up resources when your Python program terminates, (for example, disconnect from a server or database), you can use the function `register` from the module `atexit`.

You pass in a function to `register`, and the function you pass it is scheduled to run when your program terminates (even if it terminates because of an exception).

`register` can also be used as a decorator:

```
import atexit

@atexit.register
def cleanup():
    """Clean up program resources."""
    fake_db.close_connection()
    print("All cleaned up!")
```

9 – `map` with multiple arguments

The Python built-in `map` can be used with 2 or more iterable arguments.

The function beig mapped will take one argument from each iterable:

```
bases = [2, 3, 4, 2, 3, 4]
exps = [2, 2, 2, 3, 3, 3]

for num in map(pow, bases, exps):
    print(num, end=" ")
# 4 9 16 8 27 64
```

This can be more convenient to use than a list comprehension/generator expression in some situations:

```
nums = (b ** exp for b, exp in zip(bases, exps))
# vs
nums = map(pow, bases, exps)
```

For a bonus crazy use, here is how to use this to create an infinite stream of perfect squares:

```
from itertools import count, repeat

squares = map(pow, count(), repeat(2))
```

12 – Run-length encoding

The module `itertools` has a very funky iterable called `groupby`. If you're imaginative, you can use it for all sorts of things.

One possible use-case is to compute the run-length encoding of an iterable. All it takes is to go through the grouped iterable and then compute the length of each group:

```
from itertools import groupby

def run_length_encoding(iterable):
    for val, group in groupby(iterable):
        yield val, len(list(group))
```

Each group is a lazy iterable iself, so you can't use `len` directly on it. That's why you see `len(list(group))` in the code above.

Here's an example usage:

```
print(list(
    run_length_encoding("AAABOAA")
))  # [('A', 3), ('B', 1), ('O', 1), ('A', 2)]
```

42 – Bounded cache

If you have a deterministic function with no side-effects that gets called very often, consider caching its results. If said function lives in a long-running application (e.g., a web server), make sure you don't run out of memory by ensuring the cache has a maximum sizes.

You can do both of these things with the decorator `functools.lru_cache`, which accepts the cache size as an argument.

For example, `@lru_cache(1024)` in the snippet below creates a cache that saves up to 1024 different call results.

```python
from functools import lru_cache

@lru_cache(1024)
def function_to_cache(*args): ...

# Some function calls...

print(function_to_cache.cache_info().currsize)  # 35
print(function_to_cache.cache_info())
# CacheInfo(hits=12, misses=35, maxsize=1024, currsize=35)
```

You can access cache information by using the method `.cache_info` that is added to the function that gets a cache.

47 – Batching API calls

Since Python 3.12 that the module `itertools` has `batched`: it accepts an iterable and it produces batches of values from that iterable.

You can use this for all sorts of batch processing. For example, you can use `batched` to batch API calls.

Many APIs have ways to handle one request or multiple similar requests at the same time. For example, my newsletter subscriber has an API endpoint that allows me to add tags to a subscriber. However, there is a similar endpoint that allows me to do the same thing to multiple subscribers at the same time.

The second option means I hit the API fewer times, which makes my code run faster. There's less back-and-forth over the network.

Here's the pseudo-code for comparison. First, handling a single user per request:

```python
users_to_update = [...]
for user in users_to_update:
    api.add_tag(user, "some tag")
```

Now, the pseudo-code for the batch updates:

```python
from itertools import batched

users_to_update = [...]
# The API can't handle more than 50 users at a time.
for user_batch in batched(users_to_update, 50):  # <--
    api.add_tag_to_users(user_batch, "some tag")
```