Optimizing Python Code

Dan Tofan

Software Engineer, PhD

@dan_tofan | www.programmingwithdan.com

Module Overview

Caching

For vs list comprehension

Efficient iterations with generators

Fast concatenation of strings

Permission or forgiveness?

Faster functions

Optimizing numerical calculations

Interpreter-based optimizations

Risky optimizations

Caching

Computing bottlenecks

Store results in a cache
Reuse results

Network bottlenecks

Store results in a cache
Reuse results

Caching Limitations

Extra memory

No side effects

Old data

How to Use Caching

Basic approach with dictionaries

Use @lru_cache()

Use third party module (joblib)

```
# Basic approach for caching


cache = {}


def heavy_calculation(order_id):

    if order_id not in cache:
        # do the heavy work

        cache[order_id] = …


    return cache[order_id]
```

◄ Use dictionary for caching

◄ Define a function

◄ Check if the input is already cached

◄ Cache expensive result for this input data

◄ Return result from cache

Demo

Slow retrieval of order details

Use @lru_cache()

Compare performance

- Without caching
- With caching

Processing Collection Items

Input is a collection of items

Process items

Output is a new list of processed items

```
# two possible approaches

orders = [120, 150, 50]



fl = []

for o in orders:

     if o > 100:

          fl.append(o * 2)



lc = [o * 2 for o in orders if o > 100]
```

◄ **Items to process**

◄ **Using a for loop**

◄ **Using a list comprehension**

For Loop vs List Comprehension

For loop	VS	List comprehension
More flexibility		Only for creating a new list
Better for adding more logic		Great for simple logic
Lengthy		Concise
Slower for simple logic		Faster for simple logic
		Set and dictionary comprehensions

Demo

Process collection of orders

Compare performance

- For loop
- List comprehension

Generator Expressions

Lazy version of comprehensions

Avoid upfront full creation

"Just in time" values

Read lines from very large files

Limitations of Generator Expressions

Iterate only once

No random access

```
orders = [120, 150, 50]
```

◀ Orders to process

```
[o * 2 for o in orders if o > 100]
```

◀ List comprehension

```
(o * 2 for o in orders if o > 100)
```

◀ Generator expression

Generator Expression vs List Comprehension

Generator expression	VS	List comprehension
()		[]
Less flexible		Very flexible
Iterate only once		Iterate many times
Access only next item		Access any item
Very low memory		High memory

Demo

Process collection of orders

- List comprehension
- Generator expression

Compare performance

- Creation
- Access

Concatenating Strings

Small, fixed number of strings

Large, varying number of strings

How to Concatenate Strings

Using +

Using f-strings

Using join()

```
items = ['hello ', 'world']
```

◀ **Strings to join**

```
items[0] + items[1]
```

◀ **Using +**

```
f'{items[0]}{items[1]}'
```

◀ **Using f-string**

```
''.join(items)
```

◀ **Using join()**

Tradeoffs

Using +
Slow performance
Very friendly
Scalable

Using f-strings
High performance
Friendly
Not scalable

Using join()
High performance
Less friendly
Scalable

Demo

Concatenate many small strings

- Using +
- Using join()

Compare performance

What About Potential Problems?

Missing files

Missing fields

Unexpected types

Permission

Check if operation will succeed, then proceed

Use if statements

Forgiveness

Handle problems after they happen

Use try/except statements

Preventing race condition bugs

```python
class Order:
    order_id = 5

new_order = Order()
```

◄ **Create new order**

```python
if hasattr(new_order, 'order_id'):
    print(new_order.order_id)
```

◄ **Permission**

```python
try:

    print(new_order.order_id)

except AttributeError as attribute:

    print(attribute)
```

◄ **Forgiveness**

Demo

Process collection of orders

Some orders are invalid

Compare permission vs forgiveness

- Few invalid orders
- Many invalid orders

Python Functions

Typical functions

Lambda functions

Cost of function calls

```
def function():
    # more code
    other_function()
    # some more code
```

◂ **Call another function**

```
def other_function():
    print('Do this')
    print('Do that')
```

◂ **Another function**

```python
def function():
    # more code
    print('Do this')
    print('Do that')
    # some more code


def other_function():
    print('Do this')
    print('Do that')
```

◂ **Get rid of the function call**

◂ **No longer needed**

Self-sufficient Function vs Calling Other Functions

Self-sufficient function	VS	Calling other functions
Duplicate code		Clean code
Less reusable		More reusable
More difficult to maintain		Easier to maintain
Better performance		Slower performance

Demo

Create collection of orders

Compare performance

- Self-sufficient function
- Calling another function
- Lambda function

Numerical Calculations

Basic arithmetic

Matrix, vector operations

Statistical calculations

Machine learning

Numpy

Unofficial standard for scientific computing

Large ecosystem

High performance

Pandas

Relies on Numpy

Data analysis and manipulation

Tabular data

High performance

More Information

Working with Multidimensional Data Using NumPy

Janani Ravi

More Information

Pandas Fundamentals

Paweł Kordek

Demo

Get sum of squared amounts

- Using a for loop
- Using NumPy

Compare performance

Python Interpreters

Running Python code on hardware

CPython

PyPy

Cython

Jython

Pyston

CPython

Official interpreter

High portability

Use latest version

PyPy

Speed boost

Highly compatible

Lags behind CPython

Demo

Install PyPy3

Compare performance

- PyPy3
- CPython

Optimization Risks

Tradeoffs

Less maintainable code

New bugs

Much effort, small gain

Examples of Risky Optimizations

Large, self-sufficient functions

Alternative Python interpreter

Multiple assignments

```
order_subtotal = 1
order_tax = 3
order_shipping = 5
order_handling = 7
```

◄ **Individual assignments**

```
order_subtotal, order_tax,
order_shipping, order_handling = 1, 3,
5, 7
```

◄ **Multiple assignments**

Demo

Assign order details

- Multiple assignments
- Individual assignments

Compare performance

Module Summary

Caching

For vs list comprehension

Efficient iterations with generators

Fast concatenation of strings

Permission or forgiveness?

Faster functions

Optimizing numerical calculations

Interpreter-based optimizations

Risky optimizations

Up Next:

Using More Threads