Speed Up Python Code

10 Tips to Maximize Your Python Code Performance

With Sample

@miladkoohi

Swipe

1.Use list comprehension

List comprehensions are faster than for loops because they are more concise and perform the same operations in fewer lines of code.

False.py

```
list_test = []
for num in range (1, 1000):
    if num%3 == 0:
        list_test.append (i)
```

True.py

```
list_test= [num for num in range (1, 1000) if num%3 == 0]
```

2.Use multi-processing, multi-threading, or asyncio

Use multi-processing, multi-threading, or asyncio to utilize multiple CPU cores and run multiple tasks simultaneously.

false.py

```python
import requests

def fetch_url(url):
    response = requests.get(url)
    return response.text

urls = ['https://example.com', 'https://google.com', 'https://github.com']

for url in urls:
    result = fetch_url(url)
    print(f'Fetched {len(result)} bytes from {url}')
```

True.py

```python
import aiohttp
import asyncio

async def fetch_url(url):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            return await response.text()

urls = ['https://example.com', 'https://google.com', 'https://github.com']

async def main():
    tasks = [fetch_url(url) for url in urls]
    results = await asyncio.gather(*tasks)
    for url, result in zip(urls, results):
        print(f'Fetched {len(result)} bytes from {url}')

if __name__ == '__main__':
    asyncio.run(main())
```

3.Using Advanced Features Such as Decorators, Generator

Generator.py

```python
def fibonacci(n):
    a, b = 0, 1
    for _ in range(n):
        yield a
        a, b = b, a + b

for num in fibonacci(10):
    print(num)  # Output: 0 1 1 2 3 5 8 13 21 34
```

Decorators.py

```python
def log_function_call(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        print(f"Called {func.__name__} with args: {args}, kwargs: {kwargs}, Result: {result}")
        return result
    return wrapper

@log_function_call
def add(a, b):
    return a + b

result = add(5, 3)  # Output: Called add with args: (5, 3), kwargs: {}, Result: 8
```

4.Use tuple Instead of list

We can use both tuples and lists to hold a collection of values. While lists are mutable, tuples are immutable. If there is no need to update your list after its creation, consider using a tuple instead. Here are the reasons:

tuple.py

```python
# Tuples are more memory-efficient

import sys


l = ['a', 'b', 'c']
t = ('a', 'b', 'c')
print(sys.getsizeof(l))  # Output: 80
print(sys.getsizeof(t))  # Output: 64
```

5.Use built-in functions and libraries

built-in.py

```python
# Example 1: Using built-in functions for list operations
numbers = [1, 2, 3, 4, 5]
total = sum(numbers)  # Calculate the sum of elements in a list
maximum = max(numbers)  # Find the maximum element in a list
minimum = min(numbers)  # Find the minimum element in a list

# Example 2: Using built-in functions for string manipulation
text = "Hello, World!"
length = len(text)  # Get the length of a string
uppercase = text.upper()  # Convert a string to uppercase
lowercase = text.lower()  # Convert a string to lowercase
```

standard-library.py

```python
# Example: Using the datetime library for date and time operations
from datetime import datetime

current_time = datetime.now()  # Get the current date and time
formatted_time = current_time.strftime("%Y-%m-%d %H:%M:%S")  # Format date and time as a string
```

6.Concatenate strings with join

join() concatenates strings faster than + operation because + operators create a new string and then copies the old content at each step. But join() doesn't work that way.

False.py

```
concatenatedString = "Programming " + "is " + "fun."
```

True.py

```
concatenatedString = " ".join (["Programming", "is", "fun."])
```

7.Use special libraries to process large datasets

C/C++ is faster than python. So, many packages and modules have been written in C/C++ that you can use in your python programme. Numpy, Scipy and Pandas are three of them and are popular for processing large datasets.

numpy.py

```python
import numpy as np

# Create a NumPy array from a Python list
data = [1, 2, 3, 4, 5]
arr = np.array(data)

# Perform operations on the array
sum_arr = np.sum(arr)
mean_arr = np.mean(arr)
```

scipy.py

```python
from scipy import optimize

# Define a simple optimization problem
def objective(x):
    return x**2 + 5*x + 6

result = optimize.minimize(objective, x0=0)
```

pandas.py

```python
import pandas as pd

# Read a large CSV file into a DataFrame
data = pd.read_csv('large_dataset.csv')

# Perform data analysis operations
mean_value = data['column_name'].mean()
filtered_data = data[data['column_name'] > 50]
```

8. Import Properly

How do imports affect the code performance? When we use explicit imports and only import the modules and functions that we actually need, Python does not load the entire library. The fewer to be imported, the faster. For example, instead of import os, use from os import path.

Import-properly.py

```python
# Bad practice: Importing the entire module
import os

# Good practice: Importing only what's needed
from os import path

# Now, you can use path.join without loading the entire os module
file_path = path.join('folder', 'file.txt')
```

9.Use special libraries to process large datasets

Recursive functions can slow down your code because they take up a lot of memory. Instead, use iteration.

factorial_recursive.py

```python
def factorial_recursive(n):
    if n == 0:
        return 1
    else:
        return n * factorial_recursive(n - 1)
```

factorial_iterative.py

```python
def factorial_iterative(n):
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers.")
    result = 1
    for i in range(2, n + 1):  # Start from 2 to avoid multiplying by 1
        result *= i
    return result
```

9.Use special libraries to process large datasets

Recursive functions can slow down your code because they take up a lot of memory. Instead, use iteration.

factorial_recursive.py

```python
def factorial_recursive(n):
    if n == 0:
        return 1
    else:
        return n * factorial_recursive(n - 1)
```

factorial_iterative.py

```python
def factorial_iterative(n):
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers.")
    result = 1
    for i in range(2, n + 1):  # Start from 2 to avoid multiplying by 1
        result *= i
    return result
```

10.Be Careful with Global Variables

In theory, using global variables makes your code slower than using local variables. This is because when a function uses a global variable, Python needs to look up the variable in the global namespace every time the function is called. However, the performance impact of this lookup is insignificant.

The performance impact of global variables is noticeable only when we have functions that work with the global variables frequently. In this case, every modification to the global variable requires Python to look up the variable in the global namespace and update its value. The faster solution is to use local variables.

global.py

```python
global_var = 10  # This is a global variable

def modify_global_variable():
    global global_var
    global_var = 20

def use_local_variable():
    local_var = 30  # This is a local variable
    print("Inside the function: global_var =", global_var)  # Access the global variable
    print("Inside the function: local_var =", local_var)    # Access the local variable

# Call the functions
modify_global_variable()
use_local_variable()

print("Outside the function: global_var =", global_var)  # Access the global variable outside the function
# print("Outside the function: local_var =", local_var)  # This would raise an error since local_var is not accessible here
```

@miladkoohi

I have code experiences and my studies in the field of:
Python | Rust | Django | Flask | Clean codes | Clean software architecture Free software | DDD | TDD | DevOps | Data streaming

---

You can follow me and repost my content,
I am ready to receive your comments