STAT 243 Problem Set 4

Treves Li

2024-10-09

Collaboration Statement

I did not collaborate with anyone.

Question 1

Memoization of a function involves storing (caching) the values produced by a given input and then returning the cached result instead of rerunning the function when the same input is provided in the future. It can be a good approach to improve efficiency when a calculation is expensive (and presuming that sometimes the function will be run with inputs on whic it has already been used). It's particularly useful for for recursive calculuations that involve solving a subproblem in order to solve a given (larger) problem. If you've already solved the subproblem you don't need to solve it again. For example if one is working with graphs or networks, one often ends up writing recursive algorithms. (Suppose, e.g., you have a geneology giving relationships of parents and children. If you wanted to find all the descendants of a person, and you had already found all the descendants of one of the person's children, you could make use of that without finding the descendants of the child again.)

Write a decorator that implements memoization. It should handle functions with either one or two arguments (write one decorator, not two!). You can assume these arguments are simple objects like numbers or strings. As part of your solution, explain whether you need to use `nonlocal` or not in this case.

Test your code on basic cases, such as applying the log-gamma function to a number and multiplying together two numbers. These may not be realistic cases, because the lookup process could well take more time than the actual calculation. To assess that, time the memoization approach compared to directly doing the calculation. Be careful that you are getting an accurate timing of such quick calculations (see Unit 5 notes)!

I wrote a decorator function called `memoization_fun` that takes as input some other function `func`, and does the caching in a dict called `cache`.

The nested `wrapper` function takes one or two arguments–if more than two arguments are provided, then only the first two are stored as keys in the cache. The `wrapper` function then checks if the arguments are already in the cache's keys. If so, then it simply returns the cached value(s) for efficiency's sake. If not, it stores the value(s) returned by the original function in the dictionary, before also returning the cached value(s).

I used `timeit` to get accurate timings of quick calculations.

There is no need to specify `nonlocal`, since there's no modification being done on variables outside the local scope. Instead, the dictionary is just storing values.

```
def memoization_func(func):
  # Initialise dictionary to store previously computed value(s)
  cache = {}

  def wrapper(*args):
    # Check how many arguments are given
    # We only want the first two arguments to be stored as keys in the cache
    key = args if len(args) <= 2 else args[:2]

    # If the key is already in the cache, simply return it
    if key in cache:
        return cache[key]

    # Otherwise, store the result of the function in the cache
    cache[key] = func(*args)
    return cache[key]

  # Add a method to clear the cache
  wrapper.clear_cache = cache.clear

  return wrapper
```

To test my decorator, I run it on the log-gamma function and a function to multiply two numbers together. Note that I clear the cache when timing the case WITHOUT memoization, otherwise subsequent executions during the timing process would already be using the cached result from the first execution, thus making the comparison pointless.

```
import timeit
import math


# Define functions
@memoization_func
def log_gamma(x):
  return math.lgamma(x)

@memoization_func
def multiply(a, b):
  return a * b


# Run log-gamma test cases
log_gamma_input = 42

print("Running log-gamma WITHOUT memoization ...")
log_gamma_time_no_cache = timeit.timeit(
  lambda: (log_gamma.clear_cache(), log_gamma(log_gamma_input)), number=500
)
print(f"{log_gamma_time_no_cache:.20f} seconds.\n")

# Populate the cache
log_gamma(log_gamma_input)

print("Running log-gamma WITH memoization ...")
log_gamma_time_yes_cache = timeit.timeit(
  lambda: log_gamma(log_gamma_input), number=500
)
print(f"{log_gamma_time_yes_cache:.20f} seconds.\n")


# Run multiplication test cases
a = 23
b = 18

print("Running multiplication WITHOUT memoization ...")
multiply_no_cache = timeit.timeit(
  lambda: (multiply.clear_cache(), multiply(a, b)), number=500
)
print(f"{multiply_no_cache:.20f} seconds.\n")
```

```
# Populate the cache
multiply(a, b)

print("Running multiplication WITH memoization ...")
multiply_yes_cache = timeit.timeit(
  lambda: multiply(a, b), number=500
)
print(f"{multiply_yes_cache:.20f} seconds.\n")


# Compare results
print(
  f"Log-gamma speedup: \
  {round(log_gamma_time_no_cache/log_gamma_time_yes_cache, 2)}"
)

print(
  f"Multiplication speedup: \
  {round(multiply_no_cache/multiply_yes_cache, 2)}"
)
```

```
Running log-gamma WITHOUT memoization ...
0.00020398899869178422 seconds.

Running log-gamma WITH memoization ...
0.00008568100020056590 seconds.

Running multiplication WITHOUT memoization ...
0.00020129699987592176 seconds.

Running multiplication WITH memoization ...
0.00008129100024234504 seconds.

Log-gamma speedup:   2.38
Multiplication speedup:   2.48
```

The results show that memoization/caching speeds up the original functions by at least 2 times. (The speedup also depends on how fast my Jupyter session runs the code, versus how fast Quarto runs it.)

Question 2

Let's work more on how Python stores and copies objects, in particular lists.

Question 2a

Experiment with creating lists of real-valued numbers of different lengths. Try to determine how much memory is used for each reference (each element of the list), not counting the memory used for the actual values. Also consider lists of more complicated objects to see if the behavior is the same or similar.

First I generate six lists of different lengths with random, real-valued numbers. I then use `sys.getsizeof()` to get each list's size, which is comprised of references to the values stored in the list but not the actual values themselves. The amount of memory used on average for each reference can be obtained by dividing the list size by the number of list elements.

```
import random
import sys

# Define some arbitrary list lengths
lengths = [1, 10, 100, 1000, 10000, 100000]

# Create lists with arbitrary lengths
for length in lengths:
  real_list = [random.uniform(0, 100) for _ in range     (length)]

  # Get size of the list object
  list_size = sys.getsizeof(real_list)

  # Get memory for references
  element_size = list_size / length

  print(
    f"\nFor a list of length {length}:\n"
    f"{element_size:.2f} bytes is used for each element."
  )
```

```
For a list of length 1:
88.00 bytes is used for each element.

For a list of length 10:
18.40 bytes is used for each element.

For a list of length 100:
9.20 bytes is used for each element.

For a list of length 1000:
8.86 bytes is used for each element.

For a list of length 10000:
8.52 bytes is used for each element.

For a list of length 100000:
8.01 bytes is used for each element.
```

The results show that the allocated memory per reference in the list decreases as the list length increases. This is mainly due to the fixed overhead associated with the list object being spread across more elements. When a list is created, it incurs a constant memory overhead for metadata, which is more significant relative to smaller lists. As the list grows, this overhead becomes less significant per element.

We can try the code for lists of more complicated objects to see if the behaviour is similar. In this example, I use a list of tuples with three elements (a float, a string, and an int):

```python
for length in lengths:
  # Create a list of dictionaries to represent complex objects
  complex_list = [(random.uniform(0, 1), f"string_{i}", i) for i in range(length)]

  # Get size of the list object
  list_size = sys.getsizeof(complex_list)

  # Get memory for references
  element_size = list_size / length

  print(
    f"\nFor a list of length {length}:\n"
    f"{element_size:.2f} bytes is used for each tuple element."
  )
```

```
For a list of length 1:
88.00 bytes is used for each tuple element.

For a list of length 10:
18.40 bytes is used for each tuple element.

For a list of length 100:
9.20 bytes is used for each tuple element.

For a list of length 1000:
8.86 bytes is used for each tuple element.

For a list of length 10000:
8.52 bytes is used for each tuple element.

For a list of length 100000:
8.01 bytes is used for each tuple element.
```

From the above, it appears that the memory used for referencing list elements is the same regardless of whether the elements are "simple" or more "complex" objects. (This is despite the more complex "tuple" objects intuitively using up more memory for the actual element values, compared to the "simple" float objects.)

Question 2b

> Use the list `.copy` method with a list of numbers and with a list containing more complicated objects (including another list or some dictionary). Compare the behavior in terms of what happens when you modify elements at different levels of nestedness. Relate the behavior to the help information in `help(list.copy)` and to what we know about how the structure of lists.

First I create a `simple_list` with numbers, and another `complex_list` (which has a dictionary, and an inner list comprised of `simple_list`). I then make copies of both lists.

We expect that for the `simple_list`, creating a copy creates a separate object in memory. So modifying an element in the `simple_list_copy` should not affect the original `simple_list`.

However, for the `complex_list`, creating a copy will make a new outer list object, but the elements inside the copied list would still be pointing to the original objects.

We can test the above by running the `id()` function after modifying arbitrary elements in each of the simple and complex lists.

```
# Create a simple list
simple_list = [random.uniform(0, 100) for _ in range(100)]

# Create a complex list
complex_dict = [
  {'key1': random.uniform(0, 100), 'key2': f'object_{i}'}
  for i in range(100)
]
complex_list = [complex_dict, simple_list]

# Create copies of both the simple and complex list
simple_list_copy = simple_list.copy()
complex_list_copy = complex_list.copy()

# Modify an arbitrary element in the simple list
simple_list_copy[42] = 'change me'
print("\nFor the \"simple\" list:")
print(
  "ID of original element: \t\t\t\t\t"
  f"{id(simple_list[42])}"
)
```

```
print(
  "ID of copied element after modification: \t"
  f"{id(simple_list_copy[42])}"
)
print(
  "Are the two IDs the same? \t\t\t\t   ",
  id(simple_list[42]) == id(simple_list_copy[42])
)

# Modify an arbitrary element in the complex list
complex_list_copy[0][16]['key1'] = 'change me'
print("\nFor the \"complex\" list (first example using dict object):")
print(
  "ID of original element: \t\t\t\t\t"
  f"{id(complex_list[0][16]['key1'])}"
)
print(
  "ID of copied element after modification: \t"
  f"{id(complex_list_copy[0][16]['key1'])}"
)
print(
  "Are the two IDs the same? \t\t\t\t   ",
  id(complex_list[0][16]['key1']) == id(complex_list_copy[0][16]['key1'])
)

# Modify another arbitrary element in the complex list
complex_list_copy[1][80] = 'change me'
print("\nFor the \"complex\" list (second example, using inner list):")
print(
  "ID of original element: \t\t\t\t\t"
  f"{id(complex_list[1][80])}"
)
print(
  "ID of copied element after modification: \t"
  f"{id(complex_list_copy[1][80])}"
)
print(
  "Are the two IDs the same? \t\t\t\t   ",
  id(complex_list[1][80]) == id(complex_list_copy[1][80])
)
```

```
For the "simple" list:
ID of original element:                     140439902373680
ID of copied element after modification:    140439901942768
Are the two IDs the same?                   False

For the "complex" list (first example using dict object):
ID of original element:                     140440019200432
ID of copied element after modification:    140440019200432
Are the two IDs the same?                   True

For the "complex" list (second example, using inner list):
ID of original element:                     140440019202480
ID of copied element after modification:    140440019202480
Are the two IDs the same?                   True
```