Memoizacja

Memoizacja to technika optymalizacji służąca do przyspieszania wywołań funkcji przez przechowywanie ich wyników w pamięci podręcznej. Wynik funkcji może zostać przechowany tylko wtedy, gdy funkcja jest *czysta*, czyli nie ma żadnych skutków ubocznych i nie zależy od żadnego globalnego stanu (dodatkowe informacje o funkcjach czystych znaleźć można w rozdziale 8).

Prosty przykład funkcji, którą można poddać memoizacji, stanowi funkcja `sin()`, przedstawiona na listingu 10.17.

```
>>> import math
>>> _SIN_MEMOIZED_VALUES = {}
>>> def memoized_sin(x):
...     if x not in _SIN_MEMOIZED_VALUES:
...         _SIN_MEMOIZED_VALUES[x] = math.sin(x)
...     return _SIN_MEMOIZED_VALUES[x]
>>> memoized_sin(1)
0.8414709848078965
>>> _SIN_MEMOIZED_VALUES
{1: 0.8414709848078965}
>>> memoized_sin(2)
0.9092974268256817
>>> memoized_sin(2)
0.9092974268256817
>>> _SIN_MEMOIZED_VALUES
{1: 0.8414709848078965, 2: 0.9092974268256817}
>>> memoized_sin(1)
0.8414709848078965
>>> _SIN_MEMOIZED_VALUES
{1: 0.8414709848078965, 2: 0.9092974268256817}
```

Listing 10.17. Funkcja sin() poddana memoizacji

Na listingu 10.17, gdy funkcja `memoized_sin()` jest wywoływana po raz pierwszy z argumentem, który nie znajduje się w słowniku `_SIN_MEMOIZED_VALUES`, jej wartość jest wyliczana i przechowywana w tym słowniku. Jeśli ponownie wywołamy funkcję z tą samą wartością, to wynik zostanie pobrany ze słownika, zamiast wyliczony. Wartość funkcji `sin()` jest wyliczana bardzo szybko, jednak niektóre zaawansowane funkcje wymagające przeprowadzenia skomplikowanych obliczeń są bardziej czasochłonne i to w takiej sytuacji memoizacja przynosi najlepsze efekty.

Jeśli czytałeś już o dekoratorach (w przeciwnym razie zajrzyj do podrozdziału „Dekoratory i kiedy ich używać" na stronie 108), to możesz zauważyć, że jest to doskonała okazja do ich zastosowania. PyPI zawiera kilka implementacji memoizacji bazujących na dekoratorach, od bardzo prostych przypadków po bardzo złożone i kompleksowe.

Począwszy od wersji Python 3.3, moduł `functools` zawiera dekorator pamięci podręcznej LRU (*least recently used*, najdawniej użyty). Oferuje on taką samą funkcjonalność jak memoizacja, a dodatkowo ogranicza liczbę pozycji w pamięci podręcznej, usuwając najdawniej użyty element, gdy pamięć podręczna osiągnęła maksymalny rozmiar. Ten moduł oferuje również dane statystyczne m.in. liczby trafień i chybień w pamięci podręcznej (czy wartość była w niej dostępna, czy też nie). Moim zdaniem te dane statystyczne są niezbędne do implementowania tego typu pamięci podręcznej. Siła memoizacji, jak i każdej innej techniki przechowywania w pamięci podręcznej, leży w możliwości mierzenia jej użycia i użyteczności.

Listing 10.18 demonstruje, jak można wykorzystać metodę `functools.lru_cache()` do zaimplementowania memoizacji funkcji. Po udekorowaniu funkcja uzyskuje metodę `cache_info()`, którą można wywołać w celu sprawdzenia statystyk użycia pamięci podręcznej.

```
>>> import functools
>>> import math
>>> @functools.lru_cache(maxsize=2)
... def memoized_sin(x):
... return math.sin(x)
...
>>> memoized_sin(2)
0.9092974268256817
>>> memoized_sin.cache_info()
CacheInfo(hits=0, misses=1, maxsize=2, currsize=1)
>>> memoized_sin(2)
0.9092974268256817
>>> memoized_sin.cache_info()
CacheInfo(hits=1, misses=1, maxsize=2, currsize=1)
>>> memoized_sin(3)
0.1411200080598672
>>> memoized_sin.cache_info()
CacheInfo(hits=1, misses=2, maxsize=2, currsize=2)
>>> memoized_sin(4)
-0.7568024953079282
>>> memoized_sin.cache_info()
CacheInfo(hits=1, misses=3, maxsize=2, currsize=2)
>>> memoized_sin(3)
0.1411200080598672
>>> memoized_sin.cache_info()
CacheInfo(hits=2, misses=3, maxsize=2, currsize=2)
>>> memoized_sin.cache_clear()
>>> memoized_sin.cache_info()
CacheInfo(hits=0, misses=0, maxsize=2, currsize=0)
```

Listing 10.18. Sprawdzanie statystyk użycia pamięci podręcznej