Functools Module

Python 2.x 2.6–2.7 , Python 3.x 3.0–3.6

Examples

cmp_to_key

Python changed it's sorting methods to accept a key function. Those functions take a value and return a key which is used to sort the arrays.

Old comparison functions used to take two values and return -1, 0 or +1 if the first argument is small, equal or greater than the second argument respectively. This is incompatible to the new key-function.

That's where `functools.cmp_to_key` comes in:

```
>>> import functools
>>> import locale
>>> sorted(["A", "S", "F", "D"], key=functools.cmp_to_key(locale.strcoll))
['A', 'D', 'F', 'S']
```

Example taken and adapted from the Python Standard Library Documentation .

lru_cache

The `@lru_cache` decorator can be used wrap an expensive, computationally-intensive function with a Least Recently Used cache. This allows function calls to be memoized, so that future calls with the same parameters can return instantly instead of having to be recomputed.

```
@lru_cache(maxsize=None)  # Boundless cache
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

>>> fibonacci(15)
```

In the example above, the value of `fibonacci(3)` is only calculated once, whereas if `fibonacci` didn't have an LRU cache, `fibonacci(3)` would have been computed upwards of 230 times. Hence, `@lru_cache` is especially great for recursive functions or dynamic programming, where an expensive function could be called multiple times with the same exact parameters.

`@lru_cache` has two arguments

- `maxsize` : Number of calls to save. When the number of unique calls exceeds `maxsize` , the LRU cache will remove the least recently used calls.
- `typed` (added in 3.3): Flag for determining if equivalent arguments of different types belong to different cache records (i.e. if `3.0` and `3` count as different arguments)

We can see cache stats too:

```
>>> fib.cache_info()
CacheInfo(hits=13, misses=16, maxsize=None, currsize=16)
```

NOTE : Since `@lru_cache` uses dictionaries to cache results, all parameters for the function must be hashable for the cache to work.

Official Python docs for `@lru_cache` . `@lru_cache` was added in 3.2.

partial

The `partial` function creates partial function application from another function. It is used to *bind* values to some of the function's arguments (or keyword arguments) and produce a *callable* without the already defined arguments.

```
>>> from functools import partial
>>> unhex = partial(int, base=16)
>>> unhex.__doc__ = 'Convert base16 string to int'
>>> unhex('ca11ab1e')
3390155550
```

`partial()` , as the name suggests, allows a partial evaluation of a function. Let's look at at following example:

```
In [2]: from functools import partial
In [2]: from functools import partial

In [3]: def f(a, b, c, x):
   ...:     return 1000*a + 100*b + 10*c + x
   ...:

In [4]: g = partial(f, 1, 1, 1)

In [5]: print g(2)
1112
```

When `g` is created, `f` , which takes four arguments(`a, b, c, x`), is also partially evaluated for the first three arguments, `a, b, c,` . Evaluation of `f` is completed when `g` is called, `g(2)` , which passes the fourth argument to `f` .

One way to think of `partial` is a shift register; pushing in one argument at the time into some function. `partial` comes handy for cases where data is coming in as stream and we cannot pass more than one argument.

reduce

In Python 3.x, the `reduce` function already explained here has been removed from the built-ins and must now be imported from `functools` .

```
from functools import reduce
def factorial(n):
    return reduce(lambda a, b: (a*b), range(1, n+1))
```

total_ordering

When we want to create an orderable class, normally we need to define the methods `__eq()__` , `__lt__()` , `__le__()` , `__gt__()` and `__ge__()` .

The `total_ordering` decorator, applied to a class, permits the definition of `__eq__()` and only one between `__lt__()` , `__le__()` , `__gt__()` and `__ge__()` , and still allow all the ordering operations on the class.

```
@total_ordering
class Employee:

    ...

    def __eq__(self, other):
        return ((self.surname, self.name) == (other.surname, other.name))

    def __lt__(self, other):
        return ((self.surname, self.name) < (other.surname, other.name))
```

The decorator uses a composition of the provided methods and algebraic operations to derive the other comparison methods. For example if we defined `__lt__()` and `__eq()__` and we want to derive `__gt__()` , we can simply check `not __lt__() and not __eq()__` .

Note : The `total_ordering` function is only available since Python 2.7.

Syntax

Parameters

Remarks