http://pythonworld.ru/moduli/modul-functools.html#more-978

Модуль functools - сборник функций высокого уровня: взаимодействующих с другими функциями или возвращающие другие функции.

Модуль functools определяет следующие функции:

**functools.cmp\_to\_key**(func) - превращает функцию сравнения в key-функцию. Используется с инструментами, принимающие key-функции (sorted(), min(), max(), heapq.nlargest(), heapq.nsmallest(), itertools.groupby()). Эта функция в основном используется в качестве переходного инструмента для программ, преобразованных из Python 2, которые поддерживали использование функций сравнения.

Функция сравнения - функция, принимающая два аргумента, сравнивающая их и возвращающая отрицательное число, если первый аргумент меньше, ноль, если равен и положительное число, если больше. Key-функция - функция, принимающая один аргумент и возвращающая другое значение, определяющее положение аргумента при сортировке.

@**functools.lru\_cache**(maxsize=128, typed=False) - декоратор, который сохраняет результаты maxsize последних вызовов. Это может сэкономить время при дорогих вычислениях, если функция периодически вызывается с теми же аргументами.

Поскольку в качестве кэша используется словарь, все аргументы должны быть хешируемыми.

Если maxsize установлен в None, кэш может возрастать бесконечно. Также функция наиболее эффективна, если maxsize это степень двойки.

Если typed - True, аргументы функции с разными типами будут кэшироваться отдельно. Например, f(3) и f(3.0) будут считаться разными вызовами, возвращающие, возможно, различный результат.

Чтобы помочь измерить эффективность кэширования и отрегулировать размер кэша, обёрнутая функция дополняется функцией cache\_info(), возвращающая namedtuple, показывающий попадания в кэш, промахи, максимальный размер и текущий размер. В многопоточном окружении, количество попаданий и промахов приблизительно.

Также имеется функция cache\_clear() для очистки кэша.

Оригинальная функция доступна через атрибут \_\_wrapped\_\_.

from functools import lru\_cache

*# Пример получения веб-страниц*

import urllib.request

@lru\_cache(maxsize=32)

def get\_pep(num):

'Retrieve text of a Python Enhancement Proposal'

resource = 'http://www.python.org/dev/peps/pep-%04d/' % num

try:

with urllib.request.urlopen(resource) as s:

return s.read()

except urllib.error.HTTPError:

return 'Not Found'

for n in 8, 290, 308, 320, 8, 218, 320, 279, 289, 320, 9991:

pep = get\_pep(n)

print(n, len(pep))

print(get\_pep.cache\_info())

*# CacheInfo(hits=3, misses=8, maxsize=32, currsize=8)*

*# Числа Фибоначчи (попробуйте убрать lru\_cache) :)*

@lru\_cache(maxsize=None)

def fib(n):

if n < 2:

return n

return fib(n-1) + fib(n-2)

print([fib(n) for n in range(100)])

*# [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610...*

print(fib.cache\_info())

*# CacheInfo(hits=196, misses=100, maxsize=None, currsize=100)*

@**functools.total\_ordering** - декоратор класса, в котором задан один или более методов сравнения. Этот декоратор автоматически добавляет все остальные методы. Класс должен определять один из методов \_\_lt\_\_(), \_\_le\_\_(), \_\_gt\_\_(), или \_\_ge\_\_(). Кроме того, он должен определять метод \_\_eq\_\_().

Например:

@total\_ordering

class Student:

def \_\_eq\_\_(self, other):

return ((self.lastname.lower(), self.firstname.lower()) ==

(other.lastname.lower(), other.firstname.lower()))

def \_\_lt\_\_(self, other):

return ((self.lastname.lower(), self.firstname.lower()) <

(other.lastname.lower(), other.firstname.lower()))

**functools.partial**(func, \*args, \*\*keywords) - возвращает partial-объект (по сути, функцию), который при вызове вызывается как функция func, но дополнительно передают туда позиционные аргументы args, и именованные аргументы kwargs. Если другие аргументы передаются при вызове функции, то позиционные добавляются в конец, а именованные расширяют и перезаписывают.

Например:

from functools import partial

basetwo = partial(int, base=2)

basetwo.\_\_doc\_\_ = 'Convert base 2 string to an int.'

print(basetwo('10010'))

*# 18*

**functools.reduce**(function, iterable[, initializer]) - берёт два первых элемента, применяет к ним функцию, берёт значение и третий элемент, и таким образом сворачивает iterable в одно значение. Например, reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) эквивалентно ((((1+2)+3)+4)+5). Если задан initializer, он помещается в начале последовательности.

**functools.update\_wrapper**(wrapper, wrapped, assigned=WRAPPER\_ASSIGNMENTS, updated=WRAPPER\_UPDATES) - обновляет функцию-оболочку, чтобы она стала похожей на обёрнутую функцию. assigned - кортеж, указывающий, какие атрибуты исходной функции копируются в функцию-оболочку (по умолчанию это WRAPPER\_ASSIGNMENTS (\_\_name\_\_, \_\_module\_\_, \_\_annotations\_\_ и \_\_doc\_\_)). updated - кортеж, указывающий, какие атрибуты обновляются (по умолчанию это WRAPPER\_UPDATES (обновляется \_\_dict\_\_ функции-оболочки)).

@**functools.wraps**(wrapped, assigned=WRAPPER\_ASSIGNMENTS, updated=WRAPPER\_UPDATES) - удобная функция для вызова partial(update\_wrapper, wrapped=wrapped, assigned=assigned, updated=updated) как [декоратора](http://pythonworld.ru/osnovy/dekoratory.html) при определении функции-оболочки. Например:

from functools import wraps

def my\_decorator(f):

@wraps(f)

def wrapper(\*args, \*\*kwds):

print('Calling decorated function')

return f(\*args, \*\*kwds)

return wrapper

@my\_decorator

def example():

"""Docstring"""

print('Called example function')

example()

*# Calling decorated function*

*# Called example function*

print(example.\_\_name\_\_)

*# example*

print(example.\_\_doc\_\_)

*# Docstring*