Декораторы и модуль `functools`

Синтаксис использования декораторов

- Декоратор — функция, которая принимает другую функцию и что-то возвращает.

```
>>> @trace
... def foo(x):
...     return 42
...
```

- Аналогичная по смыслу версия без синтаксического сахара

```
>>> def foo(x):
...     return 42
...
>>> foo = trace(foo)
```

- Теперь понятно, что по имени `foo` будет доступно то, что вернула функция `trace`. Это и есть результат применения декоратора.
- Возвращаемый объект может быть любого типа.

“Теория” декораторов

- Декоратор `trace` выводит на экран сообщение с информацией о вызове декорируемой функции.

```
>>> def trace(func):
...     def inner(*args, **kwargs):
...         print(func.__name__, args, kwargs)
...         return func(*args, **kwargs)
...     return inner
```

- Применим его к тождественной функции

```
>>> @trace
... def identity(x):
...     "I do nothing useful."
...     return x
...
>>> identity(42)
identity (42, ) {}
42
```

- Проблема с `help` и атрибутами декорируемой функции.

```
>>> help(identity)
Help on function inner in module __main__:

inner(*args, **kwargs)
```

- Возможность глобально отключать `trace` без лишних телодвижений.
- Явное указание файла при использовании `trace`

```
>>> @trace(sys.stderr)
... def identity(x):
...     return x
```

- Использование `sys.stdout` для вывода по умолчанию.

```
>>> def identity(x):
...     "I do nothing useful."
...     return x
...
>>> identity.__name__, identity.__doc__
('identity', 'I do nothing useful as well.')
>>> identity = trace(identity)
>>> identity.__name__, identity.__doc__
('inner, None)
```

`__module__`

У любой функции в Python есть атрибут `__module__`, содержащий имя модуля, в котором функция была определена. Для функций, определённых в интерпретаторе, например:

```
>>> identity.__module__
'__main__'
```

Декораторы и help: решение в лоб

- Давайте просто возьмём и установим “правильные” значения в атрибуты декорируемой функции:

```
>>> def trace(func):
...     def inner(*args, **kwargs):
...         print(func.__name__, args, kwargs)
...         return func(*args, **kwargs)
...     inner.__module__ = func.__module__
...     inner.__name__ = func.__name__
...     inner.__doc__ = func.__doc__
...     return inner
```

- Проверим

```
>>> @trace
... def identity(x):
...     "I do nothing useful."
...     return x
...
>>> identity.__name__, identity.__doc__
('identity', 'I do nothing useful as well.')
```

- В модуле functools из стандартной библиотеки Python есть функция, реализующая логику копирования внутренних атрибутов

```
>>> import functools
>>> def trace(func):
...     def inner(*args, **kwargs):
...         print(func.__name__, args, kwargs)
...         return func(*args, **kwargs)
...     functools.update_wrapper(inner, func)
...     return inner
```

- То же самое можно сделать с помощью декоратора wraps

```
>>> def trace(func):
...     @functools.wraps(func)
...     def inner(*args, **kwargs):
...         print(func.__name__, args, kwargs)
...         return func(*args, **kwargs)
...     return inner
```

- Заведём глобальную переменную `trace_enabled` и с её помощью будем отключать и включать `trace`.

```
>>> trace_enabled = False
>>> def trace(func):
...     @functools.wraps(func)
...     def inner(*args, **kwargs):
...         print(func.__name__, args, kwargs)
...         return func(*args, **kwargs)
...     return inner if trace_enabled else func
```

- Если `trace` выключен, то результатом применения декоратора является сама функция `func` — никаких дополнительных действий в момент её исполнения производиться не будет.

Декораторы с аргументами: синтаксис

- Напоминание:

```
>>> @trace                    ≡  >>> def identity(x):
... def identity(x):              ...     return x
...     return x                  ...
                                  >>> identity = trace(identity)
```

- Для декораторов с аргументами эквивалентность сохраняется

```
>>> @trace(sys.stderr)        ≡  >>> def identity(x):
... def identity(x):              ...     return x
...     return x                  ...
                                  >>> deco = trace(sys.stderr)
                                  >>> identity = deco(identity)
```

Декораторы с аргументами: реализация

```
>>> def trace(handle):
...     def decorator(func):
...         @functools.wraps(func)
...         def inner(*args, **kwargs):
...             print(func.__name__, args, kwargs,
...                   file=handle)
...             return func(*args, **kwargs)
...         return inner
...     return decorator
...
```

Декораторы с аргументами: @with_arguments

- Можно обобщить логику декоратора с аргументами в виде декоратора with_arguments

```
>>> def with_arguments(deco):
...     @functools.wraps(deco)
...     def wrapper(*dargs, **dkwargs):
...         def decorator(func):
...             result = deco(func, *dargs, **dkwargs)
...             functools.update_wrapper(result, func)
...             return result
...         return decorator
...     return wrapper
```

- Что происходит:
  1. with_arguments принимает декоратор deco,
  2. заворачивает его во wrapper, так как deco — декоратор с аргументами, а затем в decorator.
  3. decorator конструирует новый декоратор с помощью deco и копирует в него внутренние атрибуты функции func.

Декораторы с аргументами: человечный `trace`

```
>>> @with_arguments
... def trace(func, handle):
...     def inner(*args, **kwargs):
...         print(func.__name__, args, kwargs, file=handle)
...         return func(*args, **kwargs)
...     return inner
...
>>> @trace(sys.stderr)
... def identity(x):
...     return x
```

Вопросы?

Декораторы с опциональными аргументами: наивная версия

- Почему бы просто не указать аргумент по умолчанию для параметра `handle`?

```
>>> @with_arguments
... def trace(func, handle=sys.stdout):
...     @functools.wraps(func)
...     def inner(*args, **kwargs):
...         print(func.__name__, args, kwargs,
...               file=handle)
...         return func(*args, **kwargs)
...     return inner
```

- Попробуем применить новую версию декоратора `trace` к функции `identity`

```
>>> @trace
... def identity(x):
...     return x
...
>>> identity(42)
<function trace.<locals>.inner at 0x10b3969d8>
```

Декораторы с опциональными аргументами: работающая версия

```
>>> @trace()
... def identity(x):
...     return x
...
>>> identity(42)
identity (42,) {}
42
```

Вопрос

Можно ли как-нибудь избавиться от лишних скобок?

Декораторы с опциональными аргументами: магическая версия

```
>>> def trace(func=None, *, handle=sys.stdout):
...     # со скобками
...     if func is None:
...         return lambda func: trace(func, handle=handle)
...
...     # без скобок
...     @functools.wraps(func)
...     def inner(*args, **kwargs):
...         print(func.__name__, args, kwargs)
...         return func(*args, **kwargs)
...     return inner
```

Вопрос

Зачем требовать, чтобы аргументы декоратора были только ключевыми?

- Декоратор — способ модифицировать поведение функции, сохраняя читаемость кода.
- Декораторы бывают:
  - без аргументов `@trace`
  - с аргументами `@trace(sys.stderr)`
  - с опциональными аргументами.

Практика декораторов

Использование декораторов: @timethis

```
>>> def timethis(func=None, *, n_iter=100):
...     if func is None:
...         return lambda func: timethis(func, n_iter=n_iter)
...
...     @functools.wraps(func)
...     def inner(*args, **kwargs):
...         print(func.__name__, end=" ... ")
...         acc = float("inf")
...         for i in range(n_iter):
...             tick = time.perf_counter()
...             result = func(*args, **kwargs)
...             acc = min(acc, time.perf_counter() - tick)
...         print(acc)
...         return result
...     return inner
...
>>> result = timethis(sum)(range(10 ** 6))
sum ... 0.026534789009019732
```

Использование декораторов: @profiled

```
>>> def profiled(func):
...     @functools.wraps(func)
...     def inner(*args, **kwargs):
...         inner.ncalls += 1
...         return func(*args, **kwargs)
...
...     inner.ncalls = 0
...     return inner
...
>>> @profiled
... def identity(x):
...     return x
...
>>> identity(42)
42
>>> identity.ncalls
1
```

```
>>> def once(func):
...     @functools.wraps(func)
...     def inner(*args, **kwargs):
...         if not inner.called:
...             func(*args, **kwargs)
...             inner.called = True
...     inner.called = False
...     return inner
...
>>> @once
... def initialize_settings():
...     print("Settings initialized.")
...
>>> initialize_settings()
Settings initialized.
>>> initialize_settings()
```

Вопрос

Как модифицировать декоратор `@once`, чтобы он поддерживал функции, возвращающие не **None** значения?

Использование декораторов: @memoized

- Мемоизация — сохранение результатов выполнения функции для предотвращения избыточных вычислений.
- Напишем декоратор для автоматической мемоизации “любой” функции.

```
>>> def memoized(func):
...     cache = {}
...
...     @functools.wraps(func)
...     def inner(*args, **kwargs):
...         key = args, kwargs
...         if key not in cache:
...             cache[key] = func(*args, **kwargs)
...         return cache[key]
...     return inner
```

Использование декораторов: @memoized и функция Аккермана

```
>>> @memoized
... def ackermann(m, n):
...     if not m:
...         return n + 1
...     elif not n:
...         return ackermann(m - 1, 1)
...     else:
...         return ackermann(m - 1, ackerman(m, n - 1))
...
>>> ackermann(3, 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in inner
TypeError: unhashable type: 'dict'
```

Вопрос

Что делать?

Использование декораторов: снова @memoized

- Частное решение проблемы[1]:

```
>>> def memoized(func):
...     cache = {}
...
...     @functools.wraps(func)
...     def inner(*args, **kwargs):
...         key = args + tuple(sorted(kwargs.items()))
...         if key not in cache:
...             cache[key] = func(*args, **kwargs)
...         return cache[key]
...     return inner
...
>>> ackermann(3, 4)
125
```

- Нет универсального и быстрого решения. Можно сериализовывать аргументы в строку, например, через `str` или, что более осмысленно, через `pickle`.

[1] Кстати, почему частное?

Использование декораторов: @deprecated

```
>>> def deprecated(func):
...     code = func.__code__
...     warnings.warn_explicit(
...         func.__name__ + " is deprecated.",
...         category=DeprecationWarning,
...         filename=code.co_filename,
...         lineno=code.co_firstlineno + 1)
...     return func
...
>>> @deprecated
... def identity(x):
...     return x
...
<stdin>:2: DeprecationWarning: identity is deprecated.
```

Использование декораторов: контракты pre и post

- Контрактное программирование — способ проектирования программ, основывающийся на формальном описании интерфейсов в терминах предусловий, постусловий и инвариантов.
- В Python контрактное программирование можно реализовать в виде библиотеки декораторов[2]:

```
>>> @pre(lambda x: x >= 0, "negative argument")
... def checked_log(x):
...     pass
...
>>> is_not_nan = post(lambda r: not math.isnan(r),
...                   "not a number")
>>> @is_not_nan
... def something_useful():
...     pass
...
```

[2] https://pypi.python.org/pypi/contracts

Использование декораторов: реализация @pre

```
>>> def pre(cond, message):
...     def wrapper(func):
...         @functools.wraps(func)
...         def inner(*args, **kwargs):
...             assert cond(*args, **kwargs), message
...             return func(*args, **kwargs)
...         return inner
...     return wrapper
...
>>> @pre(lambda x: r >= 0, "negative argument")
... def checked_log(x):
...     return math.log(x)
...
>>> checked_log(-42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in inner
AssertionError: negative argument
```

Использование декораторов: реализация @post

```
>>> def post(cond, message):
...     def wrapper(func):
...         @functools.wraps(func)
...         def inner(*args, **kwargs):
...             result = func(*args, **kwargs)
...             assert cond(result), message
...             return result
...         return inner
...     return wrapper
...
>>> @post(lambda x: not math.isnan(x), "not a number")
... def something_useful():
...     return float("nan")
...
>>> something_useful()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in inner
AssertionError: not a number
```

- Синтаксис Python разрешает одновременное применение нескольких декораторов.
- Порядок декораторов имеет значение:

```
>>> def square(func):
...     return lambda x: func(x * x)
...
>>> def addsome(func):
...     return lambda x: func(x + 42)
...
>>> @square
... @addsome
... def identity(x):
...     return x
...
>>> identity(2)
46
```

```
>>> @addsome
... @square
... def identity(x):
...     return x
...
>>> identity(2)
1936
```

- Декораторы в мире Python вездесущи и полезны.
- Больше примеров можно найти по ссылке[3] и практически в любой библиотеке на Python.

[3] https://wiki.python.org/moin/PythonDecoratorLibrary

Модуль functools

Модуль functools: lru_cache

- Родственник уже рассмотренного `memoized`, сохраняющий фиксированное количество последних вызовов.
- Познакомим `lru_cache` с функцией Аккермана

```
>>> @functools.lru_cache(maxsize=64)
... def ackermann(m, n):
...     # ...
...
>>> ackermann(3, 4)
125
>>> ackermann.cache_info()
CacheInfo(hits=65, misses=315, maxsize=64, currsize=64)
```

- Можно не ограничивать количество сохраняемых вызовов, тогда мы получим в точности поведение `memoized`[4]:

```
>>> @functools.lru_cache(maxsize=None)
... def ackermann(m, n):
...     # ...
...
```

[4] Почему использовать **None** в качестве значения по умолчанию для `maxsize` — плохая идея?

- С помощью `partial` можно зафиксировать часть позиционных и ключевых аргументов в функции.
- Выполнившие домашнее задание узнают в `partial` расширение функции `curry`.
- Пример:

```
>>> f = functools.partial(sorted, key=lambda p: p[1])
>>> f([("a", 4), ("b", 2)])
[('b', 2), ('a', 4)]
>>> g = functools.partial(sorted, [2, 3, 1, 4])
>>> g()
[1, 2, 3, 4]
```

Модуль `functools`: обобщённые функции[5]

- Функция `len` называется *обобщённой*, так как её реализация может быть специализирована для конкретного типа.

```
>>> len([1, 2, 3, 4])
4
>>> len({1, 2, 3, 4})
4
>>> len(range(4))
4
```

- Примеры обобщённых функций в Python:

```
>>> str([1, 2, 3, 4])
'[1, 2, 3, 4]'
>>> hash((1, 2, 3, 4))
4856967590101519O9
>>> sum([[1], [2]], [])
[1, 2]
```

- Как реализовать свою обобщённую функцию?

[5] http://python.org/dev/peps/pep-0443

Модуль functools: singledispatch

- В качестве примера реализуем функцию pack, которая сериализует объект в компактное строковое представление

```
>>> @functools.singledispatch
... def pack(obj):
...     type_name = type(obj).__name__
...     assert False, "Unsupported type: " + type_name
```

- Научим функцию pack сериализовывать числа и списки

```
>>> @pack.register(int)
... def _(obj):
...     return b"I" + hex(obj).encode("ascii")
...
>>> @pack.register(list)
... def _(obj):
...     return b"L" + b",".join(map(pack, obj))
...
>>> pack([1, 2, 3])
b'LI0x1,I0x2,I0x3'
>>> pack(42.)
AssertionError: Unsupported type: float
```

- Рассмотрим:

```
>>> sum([1, 2, 3, 4], start=0)
10
>>> (((0 + 1) + 2) + 3) + 4
10
```

- А что, если мы хотим использовать другую бинарную операцию, например, умножение?

```
>>> ((1 * 2) * 3) * 4
24
```

- Функция `reduce` обобщает логику функции `sum` на произвольную бинарную операцию.

```
>>> functools.reduce(lambda acc, x: acc * x,
...                  [1, 2, 3, 4])
24
```

- Функция `reduce` принимает три аргумента: бинарную функцию, последовательность и *опциональное* начальное значение.
- Вычисление `reduce(op, xs, initial)` можно схематично представить как:

```
>>> op(op(op(op(init, xs[0]), xs[1]), xs[2]), ...)
>>> op(op(op(xs[0], xs[1]), xs[2]), ...)
```

- Несколько примеров:

```
>>> functools.reduce(lambda acc, d: 10 * acc + int(d),
...                  "1914", initial=0)
1914
>>> functools.reduce(merge, [[1, 2, 7], [5, 6], [0]])
[0, 1, 2, 5, 6, 7]
```

- Несмотря на свою популярность в функциональных языках, в Python довольно сложно придумать **полезный** пример использования `reduce`.
- Резюме про `reduce`:
  - работает с любым объектом, поддерживающим протокол итератора;
  - работает слева направо;
  - использует первый элемент последовательности, если начальное значение не указано явно.

Модуль `functools`: резюме

- Модуль `functools` украшает будни любителя функционального программирования.
- Мы поговорили про:
  - `lru_cache`
  - `partial`
  - `singledispatch`
  - `reduce`