Домашнее задание №3

Дедлайн 1 (16 баллов): 27 сентября, 13:00
Дедлайн 2 (8 баллов): 4 октября, 13:00

Домашнее задание нужно написать на Python 3 и сдать в виде одного файла. Правило именования файла: `name_surname_03.py`. Например, если вас зовут Иван Петров, то имя файла должно быть: `ivan_petrov_03.py`.

Обратите внимание, что при выполнении задания можно использовать только то, что мы уже успели обсудить в рамках курса.

1 Следующие задания можно реализовать либо с помощью `reduce`, либо без него — руководствуйтесь здравым смыслом.

а Реализуйте функцию `union`, возвращающую объединение произвольного числа множеств.

```
>>> union({1, 2, 3}, {10}, {2, 6})
{1, 2, 3, 6, 10}
```

б Реализуйте функцию `digits`, возвращающую список цифр неотрицательного целого числа.

```
>>> digits(0)
[0]
>>> digits(1914)
[1, 9, 1, 4]
```

Пользоваться функцией `str` для реализации функции `digits` нельзя.

в Напишите функцию `lcm`, вычисляющую НОК (наименьшее общее кратное) двух и более целых чисел.

```
>>> lcm(100500, 42)
703500
>>> lcm(*range(2, 40, 8))
19890
```

г Реализуйте функцию `compose`, которая принимает две и более функции от одного аргумента, и возвращает их композицию.

```
>>> f = compose(lambda x: 2 * x, lambda x: x + 1, lambda x: x % 9)
>>> f(42)
14
>>> 2 * ((42 % 9) + 1)
14
```

2 Все декораторы из последующих заданий должны корректно работать с внутренними атрибутами функции, например, `__name__`.

а Измените декоратор `once` из лекции, чтобы он поддерживал функции, возвращающие не `None` значения.

```
>>> @once
... def initialize_settings():
...     print("Settings initialized.")
...     return {"token": 42}
...
>>> initialize_settings()
Settings initialized.
{'token': 42}
>>> initialize_settings()
{'token': 42}
```

Для функций, возвращающих `None`, декоратор должен продолжать работать.

б Измените декоратор `trace` из лекции, чтобы он выводил информацию о вызове функции, только если переданные аргументы удовлетворяют предикату.

```
>>> @trace_if(lambda x, y, **kwargs: kwargs.get("integral"))
... def div(x, y, integral=False):
...     return x // y if integral else x / y
...
>>> div(4, 2)
2
>>> div(4, 2, integral=True)
div (4, 2) {'integral': True}
2
```

в Реализуйте декоратор `n_times`. Результатом его работы должна быть функцию, вызывающая декорируемую функцию `n` раз. Возвращаемое значение декорируемой функции можно игнорировать.

```
>>> @n_times(3)
... def do_something():
...     print("Something is going on!")
...
>>> do_something()
Something is going on!
Something is going on!
Something is going on!
```

3 Существует много инструментов для сборки проектов, например, `make`, `ant`, `rake`. Обилие вариантов — следствие того, что идеального решения этой задачи ещё нет. Мы попробуем реализовать прототип похожего инструмента в виде библиотеки на Python.

а Описание сборки проекта — это набор заданий. Напишите функцию `project`, которая возвращает декоратор `register`. С помощью декоратора `register` можно запомнить (зарегистрировать) функцию как задание для сборки. Список имён всех заданий в порядке объявления в модуле должен быть доступен через метод `get_all` у декоратора `register`.

```
>>> register = project()
>>> @register
... def do_something():
...     print("doing something")
...
>>> @register
... def do_other_thing():
...     print("doing other thing")
...
>>> register.get_all()
['do_something', 'do_other_thing']
```

Добавить метод `get_all` к декоратору `register` можно через присванивание. Например, если бы мы хотели, чтобы функция `get_all` возвращала пустой список, мы бы сделали так:

```
>>> register.get_all = lambda: []
>>> register.get_all()
[]
```

Можно считать, что декоратор `register` будет применяться только к функциям без аргументов и без возвращаемого значения.

Вызов задания должен работать как вызов обычной функции.

```
>>> do_something()
doing something
>>> do_other_thing()
doing other thing
```

б Реализуйте возможность указывать зависимости между заданиями, как это показано в примере:

```
>>> @register
... def do_something():
...     print("doing something")
...
>>> @register(depends_on=["do_something"])
... def do_other_thing():
...     print("doing other thing")
...
```

Список зависимостей для задачи должен быть доступен через метод `get_dependencies`.

```
>>> do_something.get_dependencies()
[]
>>> do_other_thing.get_dependencies()
['do_something']
```

4 Можно заметить, что все объявления задач образуют ориентированный граф, в котором каждой задаче соответствует вершина, а ребро между задачами `a` и `b` проводится, если

```
>>> 'b' in a.get_dependencies()
True
```

Будем называть этот граф *графом зависимостей*. Для простоты давайте считать, что граф ацикличен.

Измените логику выполнения зарегистрированных задач таким образом, чтобы перед выполнением задачи сначала выполнились все её зависимости в порядке обратной топологической сортировки.

Для задач без зависимостей логика выполнения должна остаться без изменений.

```
>>> @register
... def do_something():
...     print("doing something")
...
>>> @register(depends_on=["do_something"])
... def do_other_thing():
...     print("doing other thing")
...
>>> do_something()
doing something
>>> do_other_thing()
doing something
doing other thing
```