Python (BSU FAMCS Fall'19)

Проект 2

Преподаватель: Дмитрий Косицин

Задание 1. Реализуйте класс `DependencyHelper`, позволяющий эффективно проверить наличие циклических зависимостей.

Реализуйте следующие методы:

- метод `add`, принимающий на вход 2 параметра – пару зависимых обьектов, где второй зависит от первого,
- оператор сложения с кортежем (парой) зависимых элементов,
- метод `update` оператор сложения с другим экземпляром `DependencyHelper`,
- метод `remove` и соответствующий ему оператор вычитания,
- метод копирования `copy`, возвращающий точную независимую копию данного объекта,
- метод `get_dependent`, принимающий в качестве аргумента некоторый элемент и возвращающий последовательность непосредственно зависимых от него элементов,
- метод `has_cycle_dependency`, проверяющий наличие циклической зависимости между обьектами,
- оператор преобразования к `bool`, возвращающий `True`, если зависимостей нет, и `False` иначе.

Пример

```
dependency_helper = DependencyHelper()
dependency_helper.add(1, 2)
dependency_helper += (2, 1)
assert not dependency_helper  # helper must find out dependend items

dependency_helper.add(2, 3)
dependency_helper.remove(2, 1)
assert dependency_helper  # no dependend items
```

Задание 2. От класса `DependencyHelper` унаследуйте класс `PriorityHelper`. В нем реализуйте метод `enumerate_priorities`, возвращающий словарь, содержащий пары обьектов и их приоритетов.

Приоритеты должны быть расставлены так, что объекты с большим приоритетом зависят от объектов с меньшим приоритетом. Значения приоритетов могут совпадать. Количество различных приоритетов постарайтесь сделать минимальным и выбирать их последовательно, начиная с нуля.

Пример

Пусть $a \leftarrow b$ (b зависит от a), $a \leftarrow c$, $d \leftarrow c$, e ни от чего не зависит. Тогда оптимальная расстановка приоритетов такова: a, d и e имеют приоритет 0, b и c – приоритет 1.

Задание 3. Примените полученные классы для построения графа зависимостей в некоторой абстрактной системе сборки.

Вам задается путь к папке. Ваша задача – обойти все вложенные папки, собрать файлы, имена которых удовлетворяют некоторому паттерну, и найти между ними зависимости.

В качестве рабочего примера будем полагать, что нам нужно найти все файлы с именем ***make***, а эти файлы в свою очередь содержат инструкции `INCLUDE` и `PEERDIR`. Первая указывает имена файлов, которые нужно собрать в этой директории. Вторая указывает директории, файлы из которых должны быть собраны раньше *всех* файлов из инструкции `INCLUDE`.

Замечание. Не завязывайтесь на имена инструкций и их действия. Предпочтительно реализовать абстрактный класс для инструкции, а в реализациях класса уже явно производить действия с графом. Добавление новых инструкций не должно быть сложным.

Примера файла *make*

```
PEERDIR(
    /path1/subpath1
    ../path2
)

INCLUDE(
    file_name1.ext1
    file_name2.ext2
)
```

Задание 4. Примените полученные классы для построения графа зависимостей файлов с **Python**-кодом.

Как и в задании выше, вам указывается путь к папке. Задача – обойти все вложенные папки, собрать файлы с **Python**-кодом, и найти между ними зависимости, которые явно прописаны в виде инструкции `import`. Для получения всех инструкций `import` можно, например, AST-дерево (другие варианты также допустимы).

Обработку директив импорта сделайте также гибкой, желательно, с помощью разных классов: сначала обрабатывайте только абсолютные импорты, далее можно добавить абсолютные импорты отдельных объектов из модулей или модулей из пакетов, далее – относительные импорты, импорты по имени модуля.

Продумайте организацию проверки существования файла или папки. Для простоты можно от нее отказаться, в последствии добавив проверку относительно корневой директории, а далее – анализировать `sys.path` и проверять, является ли модуль встроенным.

Задание 5. Предусмотрите некоторый набор юнит-тестов к вашим классам или хотя бы базовым, а также минимальные проверки аргументов на корректность.

Замечание. Все добавляемые обьекты предполагаются хешируемыми. Использовать готовые алгоритмы не допускается.

Замечание. Для реализации всего, кроме тестов и анализа файлов, допустимо использовать только стандартную библиотеку.