Функции reduce и groupby.

План урока:

1. Свертка итератора – функция `reduce`.
2. Группирование элементов – функция `groupby`.

Аннотация

На прошлом уроке мы начали знакомство с итераторами и модулем `itertools`. Хотя мы не собираемся разбирать все возможности итераторов Python, ещё две функции рассмотреть всё же стоит.

*Одна функция, которую мы рассмотрим, называется **`reduce`**, на русский язык это переводится как свёртка.*

Функция `reduce` расположена не в модуле `itertools`, а в модуле `functools`. Значит для того, чтобы ей воспользоваться, необходимо в вашей программе написать `import functools`. То, что она не находится в `itertools` – неслучайно, так как она не возвращает итератор. Напротив, она принимает итератор и, некоторым образом скомбинировав элементы, которые выдаёт этот итератор, возвращает одно единственное значение.

Итераторы сделаны "стыкующимися", так что преобразования итераторов образуют длинные цепочки. А функция `reduce` обычно завершает цепочку итераторов и возвращает итоговый результат (если только вызов `reduce` не возвращает итерируемый ответ, что возможно, но отнюдь не обязательно).

Разберемся, наконец, что такое свёртка итератора. Свёртка – функция высшего порядка, которая принимает начальное значение, итератор и некоторую бинарную операцию, а возвращает результат многократного применения этой операции к элементам итератора. Операция применяется, а результат вычисления передается как аргумент в эту же самую операцию. Функция `reduce` обновляет некоторую величину шаг за шагом, начиная с некоторого начального значения. Эта величина обновляется при получении каждого следующего элемента из итератора, когда элементы закончились, эта величина возвращается как результат работы функции.

Пусть наш итератор `iterator` возвращает элементы a_1, a_2, a_3, ..., a_n. Передадим `reduce` в качестве операции функцию двух аргументов `func(result, element)`. В качестве начального значения возьмем `init`. Последим за тем, как будет обновляться промежуточный результат:

```
functools.reduce(func, iterator, init)
```

До получения элементов из итератора промежуточный результат `result` равен `init`. После того как `reduce` получил первый элемент, промежуточный результат становится равен:

```
func(result, a1) = func(init, a1)
```

На следующем шаге:

```
result = func(result, a2) = func(func(init, a1), a2)
```

В итоге, после того как reduce получил последний элемент:

```
result = func(result, an) = func(...(func(func(init, a1), a2), ...), an)
```

Проще всего показать работу этой функции на примере. Давайте передадим в качестве операции функцию, прибавляющую очередной элемент к результату. В качестве исходного значения передадим ноль.

```
functools.reduce(lambda result, element: result + element, range(10), 0) # => 45
```

Если вы попробуете проделать все вычисления, то увидите, что мы просто посчитали сумму элементов итератора: начали с нуля и на каждом шаге прибавляли значение очередного элемента к результату. Оказывается, что сумма – это частный случай применения операции свёртки.

Совершенно аналогично можно посчитать произведение элементов, передав в reduce функцию `lambda result, element: result * element` (а в качестве начального элемента возьмём 1).

Начальный элемент указывать не обязательно. Если его не указать, то начальным значением будет первое значение итератора, а применение операции начнётся со второго элемента. Для функций суммы и произведения это будет работать как положено. Получается, что выражение **`func(... (func(func(init, a1), a2), ...), an)`** превращается в **`func(func(...func(a1, a2), ...), an)`**.

Сумма - это не единственная свёртка, которую вы уже знаете. Многие другие операции можно превратить в свёртку. Например, метод `join`:

```
values = ["картину", "корзину", "картонку"]
functools.reduce(lambda result, element: result + ", " + element, values)
# => 'картину, корзину, картонку'
```

Функцию `map` можно переписать, используя `reduce`:

```
values = ["картину", "корзину", "картонку"]
functools.reduce(lambda result, element: result + [element.upper()], values, [])
# => ['КАРТИНУ', 'КОРЗИНУ', 'КАРТОНКУ']
```

Аналогично можно написать функции **any** и **all** (если не касаться бесконечных итераторов).

Возьмём список, в котором часть элементов может помечена как отсутствующая (в списке элементы заменены на None) и попробуем с помощью reduce узнать, есть ли отсутствующие элементы в списке:

```
values = ["картину", "корзину", "картонку", None]

# all(values) -- значит, что все элементы истинны, т.е. не None и не False
# Мы используем конструкцию not not element.
# Она равна False, когда element ложный (False или None), в других случаях она равна Tr
ue
functools.reduce(lambda result, element: result and (not not element), values, True)  #
=> False

# any(values) -- значит, что хотя бы один элемент истинный
functools.reduce(lambda result, element: result or (not not element), values, False)  #
=> True
```

Для всех приведенных примеров, конечно, лучше использовать специализированные функции, а не пытаться выразить их через reduce. Операция свёртки хороша в первую очередь своей универсальностью, вы можете выразить с её помощью огромное число различных вычислений.

Задача: Свернуть к минимуму

Функцию **reduce** может быть действительно полезно использовать, когда функцию от списка значений можно записать как комбинацию более простых вызовов существующей функции пары значений. Например, если у вас есть функция, позволяющая найти пересечение двух множеств (в Python есть такая функция, см. документацию (https://docs.python.org/3/library/stdtypes.html#set) типа set (https://docs.python.org/3/tutorial/datastructures.html#sets)), то с помощью **reduce** вы сможете пересечь сколько угодно множеств.

Задача на дом: Наибольший общий делитель

Дополнительная задача: Оптимистичная лента

Группирование элементов – функция groupby.

Вторая функция, про которую мы будем говорить – **`itertools.groupby`**. Эта функция принимает итератор и группирует последовательные значения итератора, одинаковые по значению какого-либо признака. Возвращает она также итератор, который перебирает не отдельные элементы, а получившиеся группы элементов.

Это позволит нам решать задачи, которые работают не с единичным элементом, а с поднаборами коллекции. Например, мы можем взять список записей в телефонной книге и посчитать каких имён в ней больше всего. Для этого нам будет достаточно сгруппировать все записи по имени и найти самую большую группу. Мы вскоре покажем, как решать эту задачу, но перед этим нам придётся поговорить о том, что такое группа и почему группируются именно последовательные значения итератора.

Как мы уже много раз говорили, итераторы работают, последовательно перебирая элементы. Все функции, работающие с итераторами, с которыми вы имели дело, как только получают элемент, обрабатывают его и, если требуется, сразу передают полученное значение дальше. Например, когда вы работаете с функцией **`map`**, она берёт из итератора одно значение и тут же "кладёт" его в возвращаемый итератор. За счёт этого функция, которая работает с результатом функции `map`, может не ждать, пока та обработает все элементы, она получает значения из `map` по мере их вычисления. Это такой вариант потоковой обработкой данных: входящие данные – поток – обрабатываются набором последовательных преобразований, причём алгоритму обработки не требуется знать, что в потоке будет дальше. В идеальном случае необходимо знать лишь текущий элемент, в других же случаях достаточно знать текущий и предыдущие элементы.

Давайте теперь рассмотрим другой крайний случай – функцию **`sorted`**. Эта функция может получать значения из итератора, но возвращает всё равно список. Почему? Дело в том, что возвращать итератор из функции `sorted` было бы совершенно бессмысленно: мы не можем выдать ни одного значения, пока не будут прочитаны все значения итератора. Представьте, что вы хотите отсортировать список, который уже отсортирован в обратном порядке. Значит, первый элемент, который должен попасть в результат, во входящем наборе значений находится в самом конце. Таким образом потоковая обработка элементов без длительной задержки – невозможна. Кроме того, для сортировки набора значений из потока, требуется их все запомнить, что в общем случае занимает памяти как минимум столько, сколько данных было в потоке. Как вы понимаете, операция сортировки "дорого" обходится программе, но это неизбежно.

Группировка значений в потоке занимает промежуточную нишу. Если вы хотите из некоторого произвольного множества получить все значения, которые относятся к группе (для простоты будем считать, что группа содержит одинаковые значения), вам придётся прочитать все элементы до единого. Но зачастую во входящем потоке данных элементы расположены неслучайно, а заранее отсортированы. В таком случае, элементы группы идут подряд, и, чтобы их получить, достаточно взять элементы от начала группы до её конца. Раз так, мы можем сделать итератор, который перебирает группы и выдает их по-одной. Мы можем выдать группу элементов как только прочитали её до конца, т.е. в тот момент, когда началась другая группа.

На этом месте можно было бы остановиться, но проблема заключается в том, что группа может быть сколь угодно большой и, таким образом, итератор может застопориться. В то же время, мы не можем выдать группу, пока не прочитали её целиком.

Чтобы разрешить эту проблему, мы прибегнем к следующему трюку: мы перебираем элементы потока. В момент, когда в потоке начинается новая группа, мы передаём группу в результирующий итератор. Но так как мы не можем передать группу, которая ещё только началась, вместо её элементов, мы передадим итератор. Этот итератор будет перебирать элементы, пока не кончится одна группа, после чего вы получите новый итератор на следующую группу.

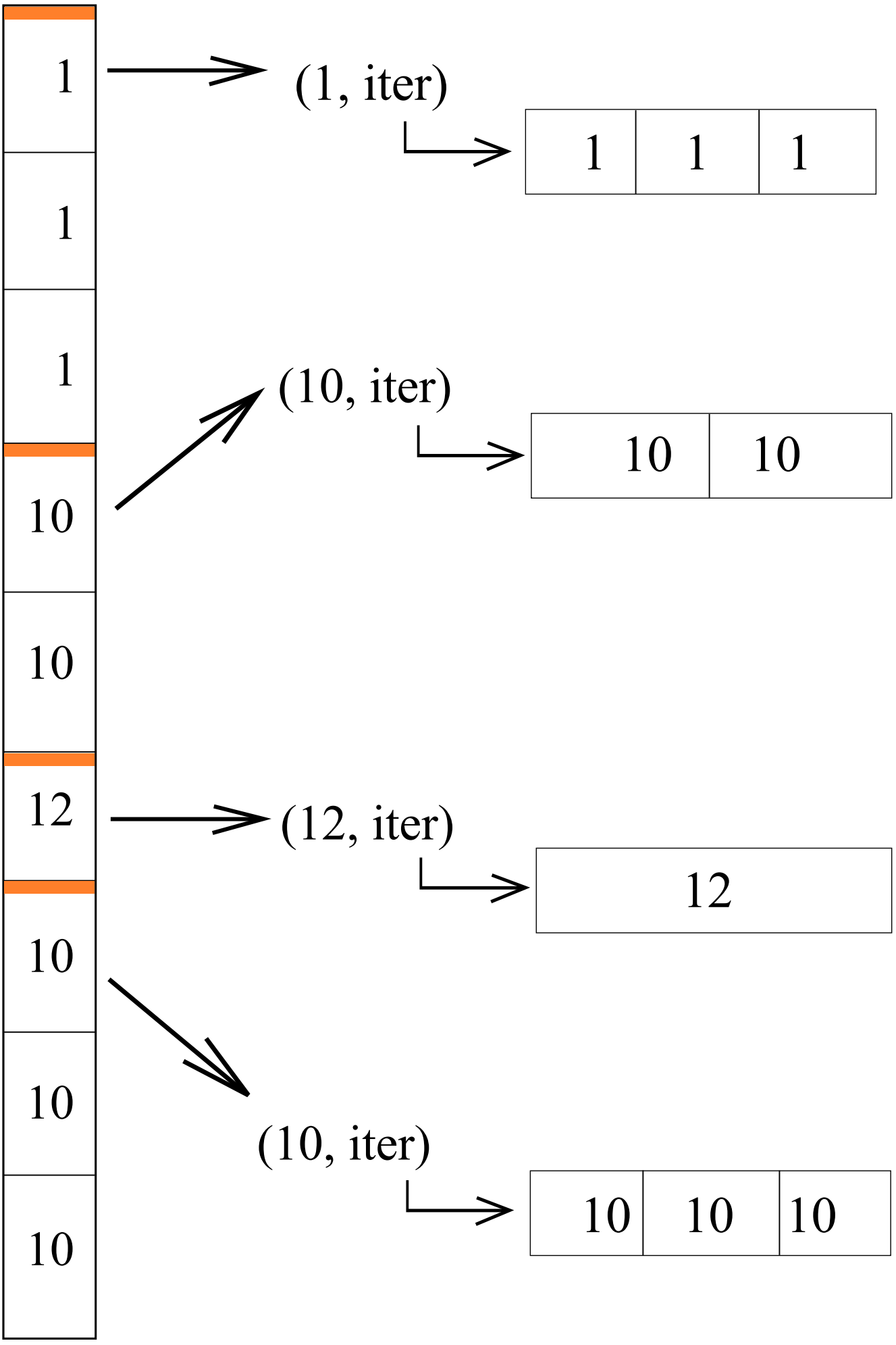
Так работает **groupby**, он создаёт итератор, состоящий из итераторов. Если быть точнее, groupby выдаёт поток кортежей, каждый кортеж состоит из значения, которое характеризует группу (соответствует всем элементам) в группе и итератора, проходящего по самим элементам группы.

Пример:

```
values = [1, 1, 1, 10, 10, 12, 10, 10, 10]

# Список кортежей (значение элементов группы, итератор группы)
print(list(itertools.groupby(values) ))
# => [(1, <itertools._grouper object at 0x...>), (10, <itertools._grouper object at 0
x...>), (12, <itertools._grouper object at 0x...>), (10, <itertools._grouper object at
 0x...>)]

# Выведем элементы каждого из внутренних итераторов
print([list(group[1]) for group in itertools.groupby(values)])
# => [[1, 1, 1], [10, 10], [12], [10, 10, 10]]
```


1
1
1
10
10
12
10
10
10
(1, iter)
1 1 1
(10, iter)
10 10
(12, iter)
12
(10, iter)
10 10 10

Обратите внимание, что поскольку мы передали не сортированный список, то у нас получились две отдельные группы десяток. Группой является множество **одинаковых** и притом **смежных** элементов. Если бы мы хотели получить группы, в которые включены все соответствующие элементы, то нам бы пришлось сначала отсортировать список:

```
print([list(group[1]) for group in itertools.groupby(sorted(values))])
# => [[1, 1, 1], [10, 10, 10, 10, 10], [12]]
```

Конечно, нам редко редко требуется группировать абсолютно одинаковые элементы, обычно мы группируем элементы по какому-либо признаку. Функцию, которая вычисляет по элементу значение группирующего признака мы передаем в качестве необязательного аргумента в функцию `groupby`.

Давайте вернёмся к примеру, с которого начинали. Пусть у нас есть список записей в телефонной книге и мы хотим найти самое частое имя. Значит первым делом, мы должны сгруппировать записи по имени, а затем посчитать размер групп. Записи будем представлять кортежами (имя, фамилия, телефон):

```
address_book = [('Андрей', 'Веселов', '235780'), ('Александр', 'Копылов', '122112'), (
'Андрей', 'Тихий', '998877')]

# Итак, отсортируем список по интересующему нас признаку и сгруппируем по нему.
# Признак - имя - является нулевым элементом записи.
# Так как он понадобится нам дважды, запишем его в переменную
key_func = lambda record: record[0]
groups = itertools.groupby(sorted(address_book, key=key_func), key_func)

# Найдём среди групп максимальную по длине группы.
# group[0] - это имя, а group[1] - это итератор по всем записям с таким именем
# sum(1 for element in iterator), как мы говорили в прошлом уроке - число элементов ите
ратора
name, group_iterator = max(groups, key = lambda group: sum(1 for record in group[1]))
print(name) # => 'Андрей'

# Но будьте осторожны! Мы присвоили значение group_iterator только для того,
# чтобы более наглядно показать значения в итоговом кортеже.
# После того, как max перебрал все элементы, group_iterator остался пустым:
print(list(group_iterator)) # => []
```

Задача: Юбилейные монеты

Задача на дом: Распределение длин

Дополнительная задача: Крупные города