Итераторы. Модуль itertools

План урока:

1. Итерируемые объекты и ленивые вычисления.
2. Итераторы и коллекции.
3. Использование итерируемых объектов в циклах и функциях преобразоваиня данных. range как итерируемый объект.
4. Поток ввода как итератор.
5. Комбинирование преобразований данных.
6. Способы перебора коллекций: `enumerate`, `zip`.
7. Проверки коллекций: `all`, `any`.
8. Функции `max/min/sorted` и использование ключа сортировки.
9. Модуль `itertools`: функции `chain`, `cycle`, `repeat`.
10. Извлечение поднабора элементов из итератора: `islice`.
11. Комбинаторные итераторы: `product`, `combinations`.

Аннотация

В Python встроено множество функций, которые помогают перебирать и комбинировать данные любыми способами. Правильно подобрав порядок преобразований ваших данных, нередко можно сложное вычисление свести к одной строке. В этом уроке мы будем изучать арсенал имеющихся инструментов и учиться их использовать. Заодно поговорим про итераторы и ленивые вычисления, позволяющие работать даже с бесконечными коллекциями объектов.

На прошлом занятии мы познакомились с лямбда-функциями и функциями высшего порядка. Вы научились пользоваться функциями `filter` и `map`, а также написали несколько собственных функций, которые принимают другие функции в качестве аргумента. Основное применение таких функций — работа с наборами (коллекциями) данных, такими как списки.

Мы можем отфильтровать список, оставив в нём только длинные слова (длина которых больше или равна, например, 7):

```
# При помощи функции filter
list(filter(lambda word: len(word) >= 7, ['Останутся', 'только',
                                          'длинные', 'слова']))
# => ['Останутся', 'длинные']

# При помощи списочных выражений
words = ['Останутся', 'только', 'длинные', 'слова']
[word for word in words if len(word) >= 7]
# => ['Останутся', 'длинные']
```

Или из списка строк получить список соответствующих чисел:

```
list(map(lambda s: float(s), ["123", "45.6", "100500"]))
# => [123.0, 45.6, 100500.0]
```

Но это лишь малая часть возможностей Python по обработке коллекций данных. В этом уроке мы узнаем, как еще упростить работу с коллекциями. При этом помните, что все инструменты, которые мы сегодня изучим, не являются незаменимыми. Для любых операций с наборами данных достаточно циклов, массивов и условного оператора. Однако специализированные функции позволят сократить десятки строк кода до одной короткой команды. Что не менее важно, при чтении кода эту команду будет гораздо проще «расшифровать», чем специально написанный «обходной путь». К тому же, при написании собственных методов обработки данных легко допустить ошибку.

Итерируемые объекты. Почему filter и map возвращают не список

Прежде чем обсуждать новые функции, нужно немного поговорить про уже изученные функции `map` и `filter`. Вы, возможно, помните, что эти функции принимают любую коллекцию (список, кортеж, строку символов и т. д.). Возвращают эти функции уже не список, а специальный объект, который можно затем передать в список, в цикл for и в некоторые другие функции. Давайте разберемся, как это работает, и почему так сделано.

Для начала поймём, почему эти функции возвращают не список. Представьте, что вы работаете с очень большим списком. Например, списком из миллиарда чисел (он занимает не меньше 4 гигабайт памяти). Если вам требуется как-то обработать набор квадратов этих чисел, есть несколько вариантов.

Первый — перебирать элементы обычным циклом `for` и отказаться от комбинирования операций, которое вы научились делать при помощи `map` и `filter`. Этот вариант, наверное, самый простой, но не слишком удобный. Особенно учитывая, что помимо `map` и `filter`, вы познакомитесь со множеством других удобных функций, работающих аналогично.

Второй вариант — сделать список квадратов, затем работать уже с ним. Это удобно, но придется потратить ещё несколько гигабайт оперативной памяти. Даже если чисел меньше миллиарда, вы вряд ли захотите, чтобы программа тратила лишнюю память.

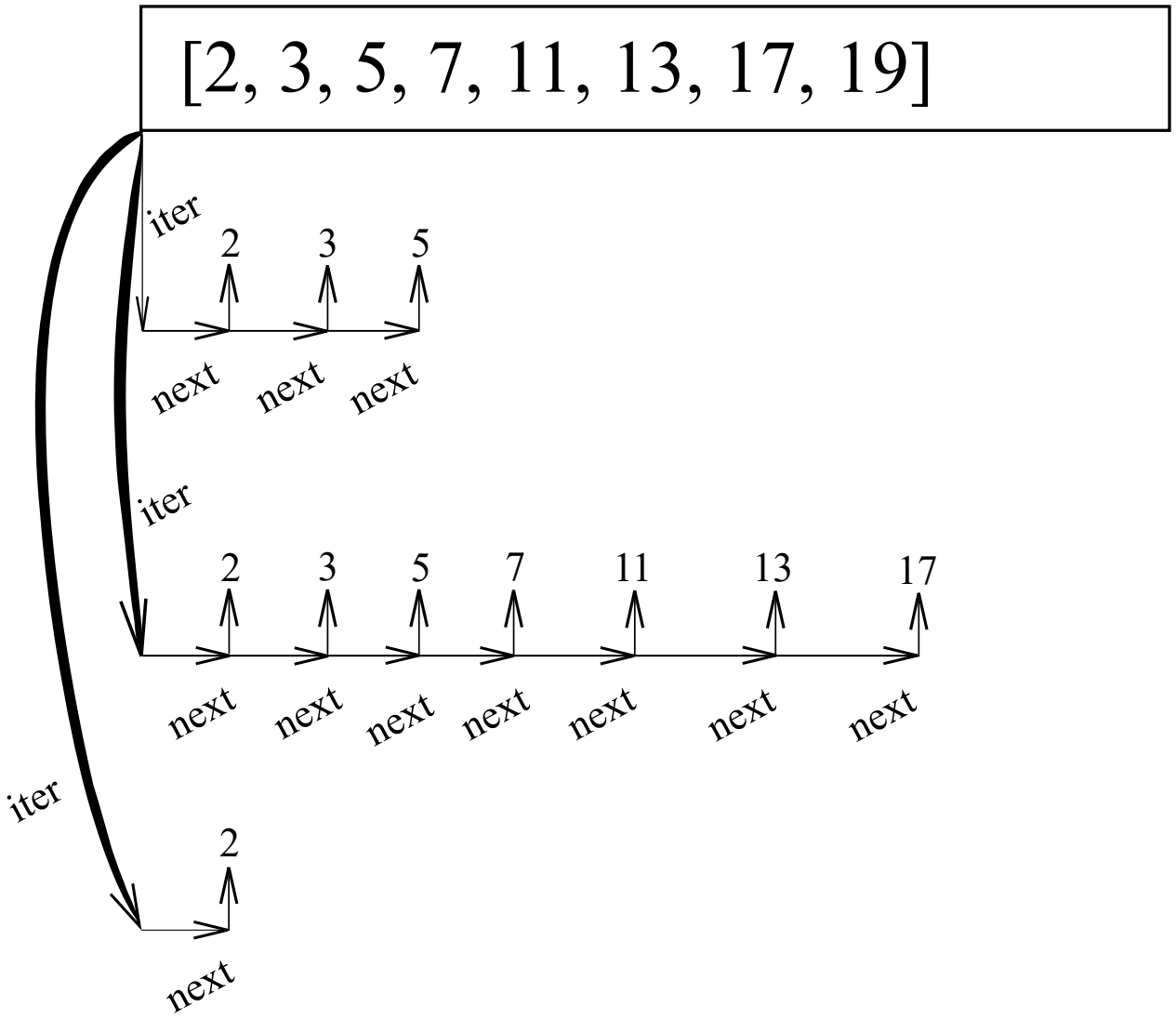
Функция `map` использует гибридный метод. Её результат позволяет перебирать не числа, а их квадраты — как мы и хотели. При этом квадраты чисел нигде не хранятся и не занимают память! Объекты, которые возвращают функции `map`, `filter` и подобные называются **итерируемыми объектами**. Это означает, что они позволяют перебирать значения по очереди и последовательно. В нашем примере функция `map` в любой момент времени хранит только то единственное число, с которым работает, а не весь миллиард квадратов исходных чисел. Вы не создаёте огромный промежуточный список и таким образом не тратите лишнюю память.

Эффект легко увидеть своими глазами. Откройте диспетчер задач и следите за потреблением памяти интерпретатором Python при запуске двух разных команд:

```
# Версия, создающая промежуточный список.
# Осторожно: при запуске этой команды, Python сначала
# занимает несколько сотен мегабайт оперативной памяти,
# а затем, когда список становится не нужен - освобождает память.

sum([x**2 for x in range(50 * 1000 * 1000)]) # => 416666654166666750000000

# Версия, работающая при помощи итератора, который
# не хранит промежуточный список.
# Она занимает минимум дополнительной памяти.

sum(map(lambda x: x**2, range(50 * 1000 * 1000))) # => 416666654166666750000000
```

Итерируемые объекты: итераторы и коллекции

Если подходить более формально, существует два типа итерируемых объектов. Первые называются итераторами. Итератор — это специальный объект, который позволяет поочередно перебирать элементы. Его можно представить как стрелочку, которая указывает на какой-то элемент коллекции и постепенно двигается по ней. Если итератор передать в функцию **next**, то функция вернёт в качестве результата следующий элемент. При этом сам итератор тоже «сдвинется» на следующий элемент. При следующем вызове функция `next` вернёт очередной элемент, и «стрелочка» снова сдвинется.


[2, 3, 5, 7, 11, 13, 17, 19]
iter
2
3
5
next
next
next
iter
2
3
5
7
11
13
17
next
next
next
next
next
next
next
iter
2
next

Второй тип итерируемых объектов — это **коллекции**. Они не итераторы сами по себе, но позволяют создать итератор. Например, список не является итератором, зато для одного и того же списка можно создать сколько угодно итераторов-стрелочек, каждая из которых будет перебирать элементы от первого до последнего. Чтобы это сделать, достаточно вызвать функцию `iter` и передать ей список в качестве аргумента. Бывает и так, что объект – сам себе итератор (с одним таким мы познакомимся).

Цикл `for` применяется к объектам второго типа. Внутри интерпретатора Python цикл `for` делает следующее:

1. По объекту создает итератор.
2. Получает из итератора объекты по одному и каждый раз передает полученный объект в выполняемый блок кода.

Благодаря этому в цикл `for` можно передать и список, и кортеж, и строку, и интервал `range`, и ещё многие другие объекты, которые имеют свои итераторы. Впрочем, эти детали не очень важны. Их полезно понимать, но пользоваться приходится редко.

Итераторы позволяют очень разным объектам «притворяться одинаковыми». Казалось бы, между строками и интервалами нет ничего общего — однако их можно итерировать, а значит, любой из этих объектов можно использовать в цикле `for`, функциях `filter/map` и многих других.

Как уже говорилось, итератор может использовать любой алгоритм выдачи значения. Элементарный итератор просто перебирает значения в списке с первого до последнего. Ещё один итератор — `range` — перебирает числа с шагом 1. Заметьте, что `range` не хранит весь набор чисел, которые будет перебирать. Он создаёт новое число только тогда, когда оно потребуется (а старые значения не хранит). Размер `range` и подобных итераторов не зависит от количества чисел, которые предполагается перебрать — ведь им нужно помнить только начало промежутка, его конец и текущий элемент. Это позволяет сделать итератор, который будет перебирать бесконечное число значений, не занимая много дополнительной памяти.

Например, итератор может перебирать все натуральные числа. Или все простые числа. Или перебирать элементы, количество которых заранее неизвестно. Если вы пробовали посчитать количество элементов, которые вернул `filter`, то могли заметить, что вызов `len(filter(condition, collection))` не работает. Теперь понятно почему: `filter` возвращает не список, а итерируемый объект.

Функция `len` не работает с большей частью итерируемых объектов, ведь длина набора может быть очень большой или даже бесконечной. Для некоторых итераторов длину можно посчитать мгновенно, не перебирая элементы. Например, итератор range имеет заранее известное число объектов. Но для других итераторов невозможно посчитать длину, не проитерировав список полностью (например, невозможно узнать число чётных элементов в списке, не проверив каждое число). Для итераторов, про которые заранее неизвестно, бесконечные они или нет, всё ещё хуже: функция `len` могла бы работать не просто долго, а вечно. От `len` такого не ожидаешь.

Если всё же требуется получить все элементы, то для этого есть уже встречавшаяся вам функция `list`. Она проходит по всем элементам итератора и собирает их в один список, который возвращает пользователю. Но `list` создаст промежуточный список. Есть более эффективный способ посчитать число элементов, возвращенных итератором, не создавая промежуточный список.

Как вы думаете, что вычисляет этот код?

```
sum(1 for value in filter(lambda x: x % 2 == 0, range(50 * 1000 * 1000)))
# => 25000000
```

Обратите внимание, что мы использовали списочное выражение как аргумент функции `sum`, но не заключали его в квадратные скобки. Мы не используем значение, которое вернул итератор, а просто добавляем единицу к числу элементов.

Большинство функций Python, которые работают с итераторами, умеют работать и с коллекциями. Поэтому слова «итерируемый объект» и «итератор» мы будем использовать как синонимы. Также, за неимением лучшего названия, мы часто будем называть итераторами функции, которые возвращают итератор (такие как `range, map, filter` и многие другие).

Поток ввода

Теперь поговорим о том, с какими итераторами работают стандартные объекты Python.

Как цикл `for` перебирает интервалы и списки с кортежами, вы уже знаете. Если проитерировать строку, вы будете получать её символы по одному:

```
for char in 'The quick brown fox jumps over the lazy dog':
    print(char)
# T
# h
# e
#
# q
# ...
```

Есть ещё один очень полезный встроенный объект: **sys.stdin**. Это — итератор так называемого «потока ввода». Поток ввода (`stdin`) — это специальный объект в программе, куда попадает весь текст, который ввёл пользователь. Потоком его называют потому, что данные хранятся там до тех пор, пока программа их не считала. Данные поступают в программу и временно «складируются» в потоке ввода, а программа может «забрать» их оттуда, например, при помощи функции `input`. В момент прочтения они пропадают из потока ввода: он хранит данные «до востребования».

`sys.stdin` — пример итератора, который невозможно перезапустить. Как и любой итератор, он может двигаться только вперёд. Но если для списка можно сделать второй итератор, который начнёт чтение с начала списка, то с потоком ввода такое не пройдёт. Как только данные прочитаны, они удаляются из потока ввода безвозвратно.

Элементы, которые выдает этот итератор — это строки, введённые пользователем. Если пользовательский ввод закончен, то итератор тоже прекращает работу. Пока пользователь не ввёл последнюю строку, мы не знаем, сколько элементов в итераторе.

Известная вам функция `input` выдает ошибку, если ввод закончился — мы скоро это увидим. Поэтому, если вы не знаете, в какой момент остановиться, то воспользоваться этой функцией не удастся. В таких случаях остаётся только работать с `sys.stdin`. Давайте немного отвлечёмся и научимся пользоваться этим итератором.

Чтобы работать с `sys.stdin`, прежде всего необходимо подключить модуль `sys` командой `import sys`. Напишем небольшую программку, которая дублирует каждую введённую пользователем строку:

```
import sys
for line in sys.stdin:
    # rstrip('\n') "отрезает" от строки line, идущий справа символ перевода строки,
    # ведь print сам переводит строку
    print(line.rstrip('\n'))
```

Если вы запустите эту программу, она может работать вечно. Чтобы показать, что ввод закончен, пользователю недостаточно нажать Enter — компьютер не знает, завершил ли пользователь работу или будет ещё что-то вводить. Вместо этого вы должны нажать Ctrl+D (если работаете в консоли Linux или IDE PyCharm) либо Ctrl+Z, затем Enter (если работаете в консоли Windows). Если вы работаете в IDE Wing, кликните правой кнопкой мыши и выберите `Send EOF`, затем нажмите Enter. Это запишет в поток ввода специальный символ `EOF (end of file)`, который отмечает конец ввода.

Мы обещали показать, что функция `input` выдаёт ошибку, если не получает ввод. Напишите простую программу:

```
x, y = input(), input()
```

Запустите программу и введите одну строку (не забудьте нажать Enter). Вместо второй строки введите EOF тем способом, которым это делается в вашей системе. Вы увидите ошибку EOFError — это означает, что `input` пытается считать данные из потока, который закончился.

В случаях, когда программе при запуске передаются данные, например, из файла, конец файла будет концом `sys.stdin` (вам не нужно нажимать Ctrl+D). Например, когда ваша программа тестируется проверяющей системой, вы знаете, что итератор закончит работу при окончании тестового ввода.

У этого итерируемого объекта есть неприятная особенность: `sys.stdin` не создает итератор, а сам является итератором. Когда он сместился на следующую строку, вернуться к предыдущей строке уже невозможно. Если вы работаете со списком, то всегда можете вернуться к началу — а символы, прочитанные из потока ввода, немедленно оттуда исчезают. Если вы не сохраните нужные строки из входного потока, то потеряете их навсегда.

Задача: Ваши комментарии

Встроенные итераторы. Комбинирование итераторов

У итераторов есть замечательная особенность: их можно комбинировать. Это позволяет вместо огромных циклов с перемешанными этапами обработки писать небольшие блоки, которые стыкуются друг с другом. В заданиях прошлого урока вам уже приходилось комбинировать итераторы, возвращаемые функциями `filter` и `map`. В этот раз вы узнаете ещё много способов комбинировать итераторы.

Начнём с итератора `enumerate`. Он решает такую задачу: представьте, что вы перебираете элементы списка при помощи итератора, но при этом хотите знать не только элемент, но и его номер. Можно завести вспомогательную переменную и изменять её в цикле, перебирающем элементы. Однако для лямбда-функций, передаваемых в функции высшего порядка, этот способ уже не работает.

`enumerate` решает эту проблему иначе: вместо элементов исходного итератора он возвращает кортежи, состоящие из элемента и его номера.

В следующем примере формируется список из подобных кортежей с использованием списочного выражения:

```
arr = ['This', 'is', 'third',' word']
print([pair for pair in enumerate(arr) ])
# => [(0, 'This'), (1, 'is'), (2, 'third'), (3, ' word')]
```

Обратите внимание на следующий пример: аргумент со звёздочкой может быть не только списком, но и итератором (как сделано в примере у функции `print`). Если перед итератором поставить звёздочку, то его элементы станут аргументами функции:

```
arr = ['This', 'is', 'third',' word']
print(*enumerate(arr))
# => (0, 'This') (1, 'is') (2, 'third') (3, ' word')
```

Давайте теперь найдём номера строк, имеющих лишь 2 буквы. Если бы мы запустили обычный `filter`, то нашли бы только сами слова без номеров. `enumerate` же передаст нам ещё и номера:

```
filtered_values = filter(lambda index_value: len(index_value[1]) == 2, enumerate(arr))
print(*filtered_values)
# => (1, 'is')
```

Теперь достаточно преобразовать пары номер-элемент в номера. Это делается тривиально:

```
map(lambda index_value: index_value[0],
    filter(lambda index_value: len(index_value[1]) == 2, enumerate(arr)))
```

Если у `enumerate` указать второй аргумент, то отсчёт начнётся не с нуля, а с этого числа. Примеры можно посмотреть в документации (https://docs.python.org/3/library/functions.html#enumerate).

Задача: Оформленные комментарии

Всегда стоит следить за тем, какие данные приходят в функцию преобразования, и какие выходят. От порядка, в котором комбинируются функции, зависит результат. Рассмотрим два варианта. Вам передается набор строк, которые вы должны пронумеровать и выбросить пустые строки. В одном случае пустые строки следует учитывать в нумерации, а в другом — пропускать.

```
# Вариант 1: сначала пронумеровали все строки, а потом отбросили пустые
lines = ['Занумеруем', '', '', 'строки']
results = filter(lambda indexed_line: len(indexed_line[1]) > 0, enumerate(lines, 1))
print(*results)
# => (1, 'Занумеруем') (4, 'строки')

# Вариант 2: сначала отбросили пустые строки, а потом пронумеровали оставшиеся
lines = ['Занумеруем', '', '', 'строки']
results = enumerate(filter(lambda line: len(line) > 0, lines), 1)
print(*results)
# => (1, 'Занумеруем') (2, 'строки')
```

Разберем ещё одну задачу. При разборе мы будем выписывать, какие данные и в какой форме получаются на каждом этапе. Задача следующая: какие по счёту високосные годы, начиная с 1582 года до 2017, имели сумму цифр, равную 9? Високосный год — это год, который делится на 4, но при этом не делится на 100, либо делится на 400.

Прежде чем писать программу, давайте разберемся, как можно было бы решить задачу, имея только лист бумаги и ручку.

- На первом этапе мы выпишем все годы от 1582 до 2017.
- Затем выберем из них високосные.
- Пронумеруем високосные годы.
- Выберем те високосные годы (уже пронумерованные), сумма цифр которых равна 9.
- Оставим от нумерованных годов только нумерацию. Готово!

Теперь дословно запишем это в коде. Мы напишем отдельную функцию для вычисления суммы цифр, а всё остальное запишем в форме комбинации итераторов.

```
def sum_digits(number):
    sum = 0
    while number != 0:
        sum += number % 10
        number //= 10
    return sum

years_range = range(1582, 2018)

is_leap_year = lambda year: (year % 4 == 0 and year % 100 != 0 or year % 400 == 0)
leap_years = filter(is_leap_year, years_range)

indexed_years = enumerate(leap_years)
selected_indexed_years = filter(lambda index_and_year: sum_digits(index_and_year[1]) ==
9, indexed_years)
year_indices = [index_and_year[0] for index_and_year in selected_indexed_years]
print(year_indices)
# => [9, 105]
```

Выполните этот код, после каждого шага выводя получившееся значение. Если вам приходится выводить итератор, то перед выводом превратите его в список, чтобы иметь возможность изучить «содержимое» получившегося итератора.

После того как поймёте и проследите каждый этап, можно немного упростить эти выражения. Хотя всю программу можно записать одной строчкой, мы так делать не будем: код будет почти невозможно прочитать. До какой степени объединять код, чтобы вам было удобно его читать, зависит от субъективных факторов. С одной стороны, очень длинная команда может с трудом восприниматься. С другой стороны, когда в программе результат каждого этапа вычислений вынесен в отдельную переменную, тяжело следить уже за этими переменными.

Таким образом, исходную задачу можно очень компактно решить только средствами итераторов:

```
leap_years = filter(lambda year: (year % 4 == 0 and year % 100 != 0 or year % 400 == 0
), range(1582, 2018))
selected_indexed_years = filter(lambda index_and_year: sum(map(int, str(index_and_year[
1]))) == 9, enumerate(leap_years))
print([index_and_year[0] for index_and_year in selected_indexed_years])
```

В рассмотренных примерах мы обходились данными из одного источника. Но бывают вычисления, в которых приходится оперировать несколькими коллекциями значений.

Например, научимся вычислять скалярное произведение двух векторов.

Каждый вектор мы можем написать в виде набора (списка) координат. Для вектора на плоскости координат две, для вектора в пространстве — три. В математических вычислениях часто приходится работать с векторами в многомерном (N-мерном) пространстве, в котором вектор описывается набором из N чисел. Скалярное произведение двух векторов вычисляется как сумма произведений соответствующих координат. Например, в двухмерном пространстве вектора с координатами (a_1, a_2) и (b_1, b_2) дают скалярное произведение $a_1b_1 + a_2b_2$.

Один способ решить задачу — перебрать все N измерений и для каждого измерения посчитать произведение элемента из первого списка на элемент из второго списка. Затем просуммировать получившиеся координаты:

```
# рассмотрим параллелепипед со сторонами 2 x 2 x 1 и вычислим скалярное произведение ме
жду
# вектором длинной диагонали и короткой "двумерной" диагонали в квадратном основании

a = [2,2,1]
b = [2,2,0]
N = len(a) # или len(b), они должны быть равны
sum(map(lambda i: a[i] * b[i], range(N))) # => 8
```

Есть и другой способ, который вообще не использует индексы элементов. Давайте выпишем два списка рядом. Пусть элементы списков a и b сгруппированы в два вертикальных набора. Если мы теперь начнем «зачитывать» элементы по горизонтали, то получим как раз N пар чисел — координаты пары векторов вдоль соответствующей оси. Для того чтобы так «перевернуть» порядок чтения, используется функция zip.

```
a    b    zip(a, b)
--------------------
 2    2     (2, 2)
 2    2     (2, 2)
 1    0     (1, 0)
```

Чтобы научиться работать с функцией zip, посчитаем с её помощью скалярное произведение:

```
sum(map(lambda coords: coords[0]*coords[1], zip(a, b))) # => 8
```

Функция `zip` вернула нам итератор, который возвращает кортежи из пар соответствующих координат. Чтобы убедиться в этом, распечатайте результат применения функции `zip` к двум спискам (не забудьте сделать из итератора список).

Задача: Длинный отрезок

В задаче «Длинный отрезок» в промежуточных итераторах приходится использовать величину, которая исходно нам не дана и не нужна в итоговом результате. В таких случаях появляется неуклюжая конструкция, при которой сначала создается сложная структура, а потом эта структура упрощается обратно.

Чтобы не писать такие конструкции, в Python есть специальный синтаксис функций вроде `min/max/sorted`. Они принимают опциональный (необязательный) параметр `key`. Параметр `key` принимает функцию, по значению которой будут сравниваться элементы. Например, пусть у нас есть набор слов, который мы хотим отсортировать:

```
words = ['мир', 'и', 'война']
```

Отсортировать слова можно различными способами. Если мы применим функцию `sorted` без аргумента `key`, то слова будут отсортированы как в словаре (это называется «лексикографически»):

```
sorted(words)
# => ['война', 'и', 'мир']
```

Теперь давайте вызовем функцию sorted следующим образом:

```
sorted(words, key = lambda s: len(s))
# => ['и', 'мир', 'война']
```

Мы указали, что в качестве ключа для сортировки должны использоваться не сами строки (встроенное в Python сравнение строк — лексикографическое), а их длины. Таким образом, мы получаем список, отсортированный по возрастанию длины слова.

Задача на дом: Словарный порядок

Помимо функции `sorted`, параметр `key` принимают функции `max` и `min`. Вызов `max(values, key)` позволяет найти значение из набора `values`, наибольшее по ключу `key`.

Задача на дом: Выгодная покупка

Уже изученная функция `zip` может принимать несколько параметров. В этом случае итератор вернет сначала кортеж из первых элементов всех переданных коллекций, затем из вторых элементов всех коллекций и так далее. Часто коллекции, которые вы хотите скомбинировать таким образом, сами находятся в отдельном списке (например, матрица — это список коллекций, представляющих значения строки). Чтобы передать все строки матрицы в функцию `zip` как отдельные строки, можно воспользоваться уже известным вам оператором «звёздочка». Проведите эксперимент и передайте в функцию `zip` матрицу, написав звёздочку и опустив её. Изучите, в чём заключаются отличия возвращаемых итератором `zip` элементов для этих случаев?

Задача на дом: Шифр сцитала

При работе с коллекциями часто приходится определять, выполняется ли некоторое условие одновременно для всех элементов коллекции или хотя бы для одного. Для этих целей существуют две функции: `all` и `any`. Первая проверяет, что все элементы переданного ей итерируемого набора значений истинны (приводятся к `True`). Вторая проверяет, что есть хотя бы один такой элемент. Эти функции могут быть полезны в комбинации с функцией map, которая для каждого элемента коллекции проверит некоторое условие и вернёт итератор, в котором будут перечисляться результаты этих проверок.

Дополнительная задача: Полумагический квадрат

Модуль itertools

Мы рассмотрели пока лишь мизерную часть возможностей Python, связанных с итераторами и их комбинированием. В стандартной библиотеке Python есть модуль `itertools`, содержащий 18 функций, которые мы ещё не изучали.Сейчас рассматривать все доступные функции незачем — это может занять не одну неделю — поэтому покажем лишь некоторые возможности. Рекомендуем вам время от времени перечитывать список функций на странице документации (https://docs.python.org/3/library/itertools.html).

Помимо встроенных функций на этой странице описано множество рецептов их применения. Со временем вы увидите, что многие задачи можно удобно переписать, используя итераторы.

Для того, чтобы работать с функциями из модуля **`itertools`**, вы должны в начале программы написать строку

```
import itertools
```

Это позволит вызывать любую функцию из этого модуля, используя её полное имя: **`itertools.<имя функции>`**.

Первым делом освоим сцепление нескольких итераторов. Мы берём несколько итераторов и делаем новый, который сначала перебирает все элементы из первого итератора, затем из второго, из третьего и так далее, пока не закончатся значения в последнем итераторе. Для этого служит функция **`itertools.chain(iter_1, iter_2, ...)`**.

Применим этот итератор для того, чтобы получить список всех дат года. Склеим этот итератор из итераторов по числам месяца, которые легко сделать методом `range`.

```
month_lengths = [31,28,31,30,31,30,31,31,30,31,30,31]
day_numbers = itertools.chain(*map(lambda length: range(1, length + 1), month_lengths))
```

В некоторых ситуациях требуется «зациклить» один итератор. Сделаем итератор часовой стрелки, который после 23 возвращается к нулю и начинает отсчет заново. Для этой цели предназначен **`itertools.cycle`**.

Если же вам нужно просто повторять какое-то значение несколько раз, можно использовать итератор **`itertools.repeat`**.

```
hours = itertools.cycle(range(24))

# Используем repeat, чтобы на протяжении 31 дня января повторять, что это первый месяц
 года и т.д.
month_lengths = [31,28,31,30,31,30,31,31,30,31,30,31]
month_numbers = itertools.chain(*map(lambda month_and_length: itertools.repeat(month_an
d_length[0], month_and_length[1]), enumerate(month_lengths, 1)))
```

Когда вы начинаете работать с бесконечными списками, возникает проблема: их работа никогда не завершается. Обычно это не то, чего мы добиваемся. Бесконечные списки часто используются как промежуточный этап работы. Для превращения бесконечного итератора в конечный существует функция **`islice`**, делающая «срез» итератора. Она принимает итератор и три параметра, определяющих начало, конец и шаг среза. Если в качестве конца среза указан `None`, то срез имеет начало, но не ограничен с конца. Срезы, конечно, можно делать не только для бесконечных итераторов, но и для конечных. Посмотрите в документации синтаксис функции `islice` и пример написания функции **`take`** из списка рецептов (https://docs.python.org/3/library/itertools.html#itertools-recipes), которая позволяет взять из итератора первые n элементов.

Дополнительные задачи: Зарубки на стене, Узорный текст

Есть большой класс итераторов, которые возвращают «комбинаторные» значения. Например, так работает итератор `itertools.product`. Результатом его работы является так называемое «декартово произведение итераторов»: для каждого значения первого итератора перебираются все значения второго. Наш итератор каждый час будет пробегать 60 минут, а затем зацикливать сутки:

```
itertools.cycle(itertools.product(range(24), range(60)))
```

Попробуйте зациклить отдельно минуты и отдельно часы, прежде чем перемножать их. Что получается в итоге?

Задача: Колода карт

Декартово произведение — самый простой комбинаторный итератор. Разберем ещё один: **`itertools.combinations`**. Он берёт итератор и некоторое число `r`, а затем выдаёт набор всех возможных комбинаций из `r` элементов (пришедших из переданного итератора). Например, `itertools.combinations(range(10), 2)` переберет все возможные пары различных цифр. При этом кортежи с переставленными элементами не появляются в результатах. Например, мы получим элемент (2, 7), но не получим элемент (7, 2), так как это та же самая комбинация элементов итератора.

Задача на дом: Карточные расклады

Дополнительная задача: Дислексия

На следующем занятии мы обсудим еще один тип итераторов, позволяющий сгруппировать элементы по некоторому признаку. Также немного поговорим про функции, которые могут и не могут работать в потоковом режиме.