

Урок 4. Импорт, модули и полезные ВОЗМОЖНОСТИ ЯЗЫКА

Урок посвящён инструментам, которые позволят сделать ваш код более лаконичным, упростить решение многих стандартных задач. В уроке описаны возможности ряда дополнительных модулей, используемых при написании алгоритмов. Приведены особенности механизма запуска скриптов с параметрами и получения доступа к ним из кода программы. Использование представленных инструментов относится к более продвинутому стилю программирования и повышает статус разработчика.

Оглавление

[Импортирование в Python](#)

[Импорт модуля из стандартной библиотеки](#)

[Использование инструкции from](#)

[Создание собственного модуля](#)

[Запуск скрипта с параметрами](#)

[Генераторы списков и словарей](#)

[Генераторы списков](#)

[Генераторы словарей и множеств](#)

[Модуль random как генератор псевдослучайных чисел](#)

[Генерация целых случайных чисел](#)

[Генерация дробных случайных чисел](#)

[Конструкция yield](#)

[Модуль functools](#)

[Функция reduce\(\)](#)

[Функция partial\(\)](#)

[Модуль itertools](#)

[Функция count\(\)](#)

[Функция cycle\(\)](#)

[Модуль math](#)

[Практическое задание](#)

[Дополнительные материалы](#)

[Используемая литература](#)

На этом уроке студент:

1. Узнает, как импортировать и использовать в своих программах встроенные модули.
2. Создаст собственные модули и подключать их к программам.
3. Научится передавать необходимые параметры при запуске скриптов.
4. Создаст списки, множества и словари с помощью генераторов.
5. Научится использовать модуль random для генерации целых и дробных случайных чисел.
6. Узнает, для чего предназначена конструкция yield.
7. Познакомится с возможностями модулей functools, itertools и math.

Импортирование в Python

«Импорт» и «модуль» — понятия пока незнакомые. Но о них нужно поговорить, поскольку функция вызывается не только в том файле, где она написана. Она может быть импортирована из другого файла с Python-кодом, называемого модулем.

Итак, модуль в Python — это файл с кодом, то есть некая программа, которую можно связать с другой. Есть встроенные модули, которые можно импортировать из стандартной библиотеки, и реализованные самим разработчиком. Благодаря модульному принципу программ мы можем связывать модули друг с другом и импортировать из них функции и классы для последующего использования.

Импорт модуля из стандартной библиотеки

Для этого применяется оператор **import**, за которым следует название модуля. С помощью одной инструкции импорта можно подключить к программе сразу несколько модулей. Но это ухудшает читаемость кода и не соответствует соглашениям PEP-8. Поэтому импортировать следует каждый модуль отдельно.

Рассмотрим применение оператора `random` с применением модулей **random** и **time**.

Пример:

```
import time
import random
print(time.time())
print(random.random())
```

Результат:

```
1563440619.2266152
0.7303585873639512
```

После импорта модуля его имя можно использовать как переменную, через которую доступны параметры и функции модуля.

Использование инструкции `from`

В примере, рассмотренном выше, импортируются модули целиком. Можно импортировать только определённые объекты модуля:

Пример:

```
from time import time
from random import random
print(time())
print(random())
```

Результат:

```
1563441483.3917782
0.5331559021496495
```

Создание собственного модуля

Отметим ещё раз, что, создавая файл с программным кодом на Python (с расширением **.py**), вы фактически воплощаете модуль, в котором можно определить переменные, функции и классы. Создадим файл-модуль **my_functions.py** и определим в нём две функции.

Пример:

```
def show_msg():
    print("Приветствие!")

def simple_calc():
    x = int(input("Введите значение x: "))
    return x ** 2 - 1
```

Теперь в директории с файлом **my_functions.py** создадим ещё один файл, например, **main.py** и выполним подключение созданного ранее модуля **my_functions.py**.

Пример:

```
import my_functions

my_functions.show_msg()
print(my_functions.simple_calc())
```

Результат:

```
Приветствие!  
Введите значение x: 4  
15
```

Можно записать по-другому:

```
from my_functions import show_msg  
from my_functions import simple_calc  
  
show_msg()  
print(simple_calc())
```

Запуск скрипта с параметрами

Выполняя запуск скриптов, пользователь зачастую должен передавать в программу некоторые данные, необходимые для выполнения скрипта. Запрашивать их у пользователя можно интерактивно, в процессе работы скрипта. Для этого применяется функция **input()**, которая отвечает за получение данных от пользователя и их сохранение в переменных.

Есть и другое решение, суть которого заключается в передаче данных в скрипт прямо в момент его запуска. Этот механизм называется запуском скрипта с параметрами.

Рассмотрим работу этого механизма на примере. Создадим простой файл-модуль, например, с именем **script_params_test.py**, и добавим в него несколько простых инструкций:

Пример:

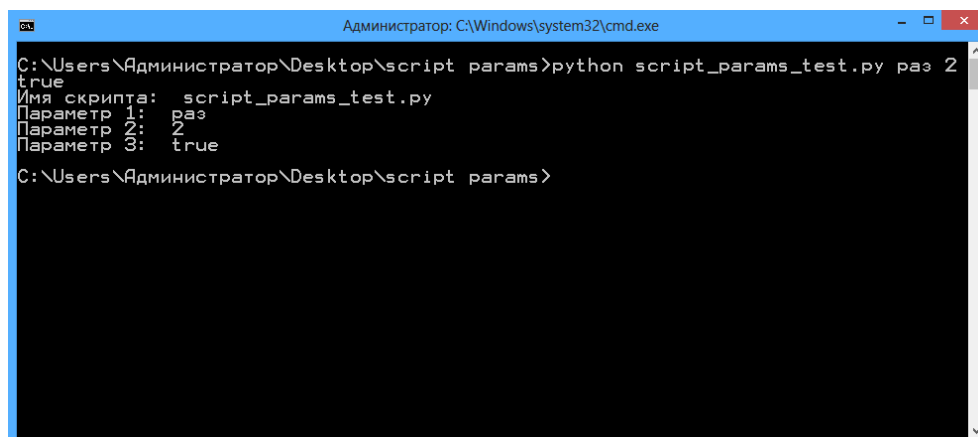
```
from sys import argv  
  
script_name, first_param, second_param, third_param = argv  
  
print("Имя скрипта: ", script_name)  
print("Параметр 1: ", first_param)  
print("Параметр 2: ", second_param)  
print("Параметр 3: ", third_param)
```

Скрипт небольшой, но позволит отразить возможности передачи данных в программу. Для его запуска нужно вызвать командную строку — желательно из директории расположения скрипта — и запустить команду:

Пример:

```
python script_params_test.py paz 2 true
```

Результат:

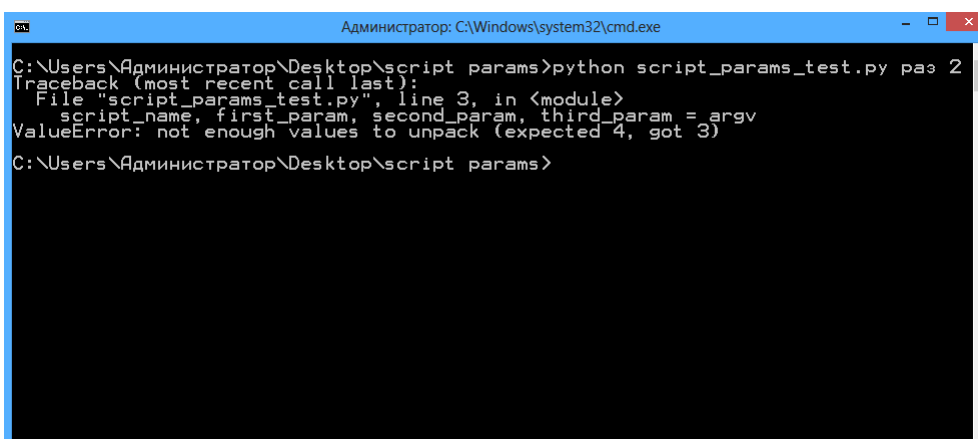


```
Администратор: C:\Windows\system32\cmd.exe
C:\Users\Администратор\Desktop\script params>python script_params_test.py paz 2
true
Имя скрипта: script_params_test.py
Параметр 1: paz
Параметр 2: 2
Параметр 3: true
C:\Users\Администратор\Desktop\script params>
```

В этом примере мы передали три параметра, отобразили их значения и сделали вывод имени скрипта. Теперь разберёмся с кодом подробнее.

Первая строка отвечает за импорт списка аргументов командной строки, переданных скрипту (**sys.argv**). В следующей — осуществляется распаковка содержимого списка **argv** в переменные. Мы как бы говорим интерпретатору Python, что он должен взять данные из списка **argv** и последовательно связать извлекаемые данные с каждой из переменных. Эти переменные указаны с левой стороны выражения. Далее мы можем выполнять нужные операции с представленными переменными.

Первый переданный параметр — имя скрипта. В качестве других параметров мы можем указать любые другие значения, отличные от примера. Но число этих значений должно совпадать с числом переменных в левой части выражения. Если, например, для этого скрипта попытаться передать два параметра вместо трёх, появится сообщение об ошибке:



```
Администратор: C:\Windows\system32\cmd.exe
C:\Users\Администратор\Desktop\script params>python script_params_test.py paz 2
Traceback (most recent call last):
  File "script_params_test.py", line 3, in <module>
    script_name, first_param, second_param, third_param = argv
ValueError: not enough values to unpack (expected 4, got 3)
C:\Users\Администратор\Desktop\script params>
```

Сообщение об ошибке возникает из-за передачи в скрипт недостаточного числа параметров. Строка с **ValueError** сообщает, что, согласно логике скрипта, необходимо передать 4 значения вместо указанных трёх.

Используя подобный механизм, важно помнить, что передаваемые в скрипт параметры — строковые. Если параметры предполагается использовать дальше в программе, их нужно преобразовать в нужный тип данных.

Генераторы списков и словарей

Это механизм, миссия которого — быстрое создание и заполнение списков и словарей в Python. Генераторы предполагают использование итерируемого объекта. На его базе формируется новый список и выражение. Второе призвано выполнить с извлечёнными из итерируемого объекта элементами некоторые операции перед их включением в итоговый список.

Генераторы списков

Генераторы — это пример так называемого синтаксического сахара в языке программирования Python. Это возможность использования таких инструкций кода, которые не меняют поведения программы. Они делают конструкции на Python более понятными.

Пример:

```
my_list = [2, 4, 6]
new_list = [el+10 for el in my_list]
print(f"Исходный список: {my_list}")
print(f"Новый список: {new_list}")
```

Результат:

```
Исходный список: [2, 4, 6]
Новый список: [12, 14, 16]
```

В приведённом примере функцию генератора выполняет выражение: **el+10 for el in my_list**, где **my_list** — итерируемый объект. Из него в цикле **for** поочерёдно извлекаются элементы. Перед инструкцией **for** указывается действие, которое выполняется над элементом перед добавлением его в новый список.

Важно! Генератор создаёт новый список, а не изменяет текущий.

Пример:

```
my_list = [2, 4, 6]
print(f"Исходный список: {my_list}")
new_list = []
for el in my_list:
    new_list.append(el + 10)
```

```
print(f"Новый список: {new_list}")
```

Результат:

```
Исходный список: [2, 4, 6]  
Новый список: [12, 14, 16]
```

В цикле **for** возможен перебор не только элементов списка, но и строк файла.

Пример:

```
lines = [line.strip() for line in open('text.txt')]  
print(lines)
```

Результат:

```
['stroka_1', 'stroka_2', 'stroka_3']
```

В генератор допускается добавление условия.

Пример:

```
my_list = [10, 25, 30, 45, 50]  
print(my_list)  
new_list = [el for el in my_list if el % 2 == 0]  
print(new_list)
```

Результат:

```
[10, 25, 30, 45, 50]  
[10, 30, 50]
```

Допустимо использовать вложенные циклы.

Пример:

```
str_1 = "abc"  
str_2 = "d"  
str_3 = "efg"  
sets = [i+j+k for i in str_1 for j in str_2 for k in str_3]
```



```
print(sets)
```

Результат:

```
['ade', 'adf', 'adg', 'bde', 'bdf', 'bdg', 'cde', 'cdf', 'cdg']
```

Обратите внимание на следующий пример:

```
my_tuple = (2, 4, 6)
new_obj = (el+10 for el in my_tuple)

print(new_obj)
```

Результат:

```
<generator object <genexpr> at 0x0000008E23521138>
```

Здесь мы используем генераторное выражение для элементов кортежа, но в результате получаем объект-итератор. Такой результат связан с использованием круглых скобок в генераторном выражении. Если в этом примере заменить кортеж на список, результат будет идентичный (объект-итератор).

Пример:

```
my_tuple = [2, 4, 6]
new_obj = (el+10 for el in my_tuple)

print(new_obj)
```

Результат:

```
<generator object <genexpr> at 0x0000003E13BB9620>
```

Генераторы словарей и множеств

Если в конструкции, определяющей генератор, вместо квадратных скобок указать фигурные, то результатом работы генератора будет словарь.

Пример:

```
my_dict = {el: el*2 for el in range(10, 20)}  
print(my_dict)
```

Результат:

```
{10: 20, 11: 22, 12: 24, 13: 26, 14: 28, 15: 30, 16: 32, 17: 34, 18: 36, 19: 38}
```

Генератор для множеств отличается незначительно:

Пример:

```
my_set = {el**3 for el in range(5, 10)}  
print(my_set)
```

Результат:

```
{512, 343, 216, 729, 125}
```

Модуль random как генератор псевдослучайных чисел

Модуль содержит специальные функции для генерации целых и дробных чисел. Рассмотрим использование этих функций на примерах.

Генерация целых случайных чисел

Применяются функции `randint()` и `randrange()`. Первая — самая простая в использовании. Она принимает два аргумента — нижняя и верхняя границы целочисленного диапазона, из которого выбирается число.

Пример:

```
import random  
print(random.randint(0, 10))
```

Результат:

```
7
```

Для функции **randint()** значения и нижней, и верхней границы входят в диапазон, из которого определяется число.

Можно работать с функцией напрямую, импортируя из модуля.

Пример:

```
from random import randint
print(randint(0, 10))
```

Результат:

```
10
```

Левая граница всегда должна быть меньше правой. Допускается использование отрицательных чисел для определения границ диапазона.

Пример:

```
from random import randint
print(randint(-100, -10))
```

Результат:

```
-78
```

Функция **randrange()** устроена сложнее. Она может принимать от одного до трёх аргументов.

1. Один аргумент — возвращается случайное число от 0 до переданного аргумента. При этом сам аргумент в диапазон не включается.

Пример:

```
from random import randrange
print(randrange(10))
```

Результат:

```
5
```

2. Два аргумента — возвращается случайное число в указанном диапазоне. При этом верхняя граница в диапазон не включается.

Пример:

```
from random import randrange
print(randrange(10, 20))
```

Результат:

```
17
```

3. Три аргумента. Первые два — нижняя и верхняя границы, третий — шаг. Например, для функции **randrange(20, 30, 3)** случайное число выбирается из чисел 20, 23, 26, 29.

Пример:

```
from random import randrange
print(randrange(20, 30, 3))
```

Результат:

```
26
```

Генерация дробных случайных чисел

Такие числа называются вещественными, или числами с плавающей точкой. Самый простой способ получить вещественное число — применить функцию **random()** без параметров. Результат её работы — число с плавающей точкой от 0 до 1, не включая верхнюю границу диапазона.

Пример:

```
from random import random
print(random())
```

Результат:

```
0.7745718967220968
```

Для генерации вещественного числа в других пределах можно воспользоваться следующим приёмом:

Пример:

```
from random import random
print(random() * 10)
```

Результат:

```
6.369620932985977
```

При этом генерируется вещественное число от 0 до указанного целого. Само целое число в диапазон не входит.

Чтобы нижняя граница отличалась от нуля, нужно число, генерируемое функцией **random()**, умножить на разность верхней и нижней границ, и прибавить нижнюю.

Пример:

```
from random import random
print(random() * (10 - 4) + 4)
```

Результат:

```
7.913607590966955
```

В этом примере результат выполнения функции **random()** умножается на 6. В результате выходит число от 0 до 6. Прибавляем 4 и получаем число от 4 до 10.

Основные функции модуля **random** представлены в таблице:

Функции	Назначение
.random()	Возвращает псевдослучайное число от 0.0 до 1.0
uniform(<Начало>, <Конец>)	Возвращает псевдослучайное вещественное число в указанных пределах
randint(<Начало>, <Конец>)	Возвращает псевдослучайное целое число в указанных пределах
choice(<Последовательность>)	Возвращает случайный элемент из любой последовательности (строки, списка, кортежа)

<code>randrange(<Начало>, <Конец>, <Шаг>)</code>	Возвращает случайно выбранное число из последовательности
<code>shuffle(<Список>)</code>	Перемешивает последовательность элементов

В таблице приводится только часть функций. С полным списком можно ознакомиться по [ссылке](#).

Конструкция `yield`

Использование конструкции **`yield`** тесно связано с понятием генератора. Это итерируемый объект, который можно использовать один раз, так как при использовании генератора значения не хранятся в памяти. Они формируются в процессе обращения к ним, по мере запроса.

Пример:

```
generator = (param * param for param in range(5))

for el in generator:
    print(el)
```

Результат:

```
0
1
4
9
16
```

Важно, что пройти по генератору можно только один раз, данные в памяти не хранятся. При повторной попытке возникнет ошибка. Например, можно попытаться получить следующее значение с помощью функции **`next()`**.

Пример:

```
generator = (param * param for param in range(5))

for el in generator:
    print(el)

print(next(generator))
```

Результат:

```
StopIteration
```

Оператор **yield** по назначению похож с оператором **return**, но возвращает генератор вместо значения.

Пример:

```
def generator():
    for el in (10, 20, 30):
        yield el

g = generator()
print(g)

for el in g:
    print(el)
```

Результат:

```
<generator object generator at 0x000000C64E181138>
10
20
30
```

Такой механизм может быть полезен в том случае, когда функция возвращает большой объём данных. Но использовать их нужно только единожды. При вызове функции с оператором **yield** функция не выполняется. Она возвращает объект-генератор, с которым далее можно выполнять нужные действия.

Модуль **functools**

Это специализированный модуль высокого порядка. Его также называют функциями, которые взаимодействуют с другими функциями и возвращают их. Для начала изучим только часть функций модуля **functools**.

Функция **reduce()**

Применяет указанную функцию к некоторому набору объектов и сводит его к единственному значению.

Пример:

```
from functools import reduce
```

```
def my_func(prev_el, el):
    # prev_el - предыдущий элемент
    # el - текущий элемент
    return prev_el + el

print(reduce(my_func, [10, 20, 30]))
```

Результат:

60

Функция partial()

Позволяет создать новую функцию с частичным указанием передаваемых аргументов.

Пример:

```
from functools import partial

def my_func(param_1, param_2):
    return param_1 ** param_2

new_my_func = partial(my_func, 2)
print(new_my_func)
print(new_my_func(4))
```

Результат:

16

В этом примере создана простая функция, возвращающая результат выполнения операции с параметрами. Далее создаётся новый экземпляр функции **partial**, в которую передаётся экземпляр исходной функции и параметр.

Модуль itertools

Содержит итераторы, выполняющие бесконечный процесс итерирования. Это требует условия разрыва итераторов, чтобы избежать бесконечного цикла. Модуль включает широкие возможности, но мы пока рассмотрим только две его функции.

Функция count()

Это итератор, возвращающий равномерно распределённые переменные с числа, переданного как стартовый параметр. Допускается указывать значения шага.

Пример:

```
from itertools import count

for el in count(7):
    if el > 15:
        break
    else:
        print(el)
```

Результат:

```
7
8
9
10
11
12
13
14
15
```

В примере импортируется функция **count()** из модуля **itertools**, и создаётся цикл **for**. В скрипт добавляется условная проверка, разрывающая цикл при превышении итератором значения 15. Иначе выводится текущее значение итератора. Результат начинается со значения 7, так как оно определено в качестве стартового.

Функция cycle()

Это функция, создающая итератор для формирования бесконечного цикла набора значения.

Пример:

```
from itertools import cycle

c = 0
for el in cycle("ABC"):
    if c > 10:
        break
    print(el)
    c += 1
```

Результат:

```
A
B
C
A
B
C
A
B
C
A
B
```

В этом примере создаётся цикл **for** для бесконечного заикливания букв А, В, С. Но создавать бесконечный цикл — плохая идея, поэтому дополнительно реализован счётчик для разрыва цикла.

Для выполнения операций перемещения по итератору применяется функция **next**.

Пример:

```
from itertools import cycle

progr_lang = ["python", "java", "perl", "javascript"]
iter = cycle(progr_lang)

print(next(iter))
print(next(iter))
print(next(iter))
print(next(iter))
print(next(iter))
print(next(iter))
```

Результат:

```
python
java
perl
javascript
python
java
```

Здесь создаётся список нескольких языков программирования, которые передаются по циклу. Далее новый итератор сохраняется в качестве переменной. Она передаётся следующей функции. При каждом вызове функции возвращается очередное значение в итераторе. Этот итератор бесконечный, поэтому ограничений на число вызовов **next()** нет.

Основные функции модуля **itertools** представлены в таблице:

Функции	Назначение
count (<Начало>, <Шаг>)	Возвращает равномерно распределённые переменные, начиная с числа — стартового параметра. Можно указать параметр шага.
cycle (<Итерируемый объект>)	Итератор, создающий бесконечный цикл поочерёдного вывода неких символов или чисел.
repeat (<Объект>, <Количество повторений>)	Итератор, осуществляющий повторение объекта, переданного в качестве первого параметра в функцию.
combinations (<Объект>, <Количество значений>)	Функция комбинирования элементов последовательности. Принимает два аргумента: объект и количество значений, которые должны присутствовать в каждой комбинации.
combinations_with_replacement (<Объект>, <Количество значений>)	Модифицированный вариант предыдущей функции. Предоставляет программе возможность делать выборку из отдельных элементов с учётом их порядка. Комбинации могут состоять из повторяющихся элементов.
permutations (<Объект>, <Количество значений>)	Сходна с предыдущей функцией, но в текущей не допускается размещение идентичных элементов в одной комбинации.
product (<Массив данных>)	Принимает в качестве параметра массив данных, объединяющий несколько групп значений. Позволяет получить из введённого набора чисел и символов новую совокупность групп во всех возможных вариациях.

В таблице приводится только часть функций. С полным списком можно ознакомиться по [ссылке](#).

Модуль math

Предоставляет многочисленные функции для работы с числами:

Функции	Назначение
ceil (N)	Округлить число N до ближайшего большего числа
fabs (N)	Определить модуль числа N
factorial (N)	Найти факториал числа N
floor (N)	Округлить число вниз
fmod (a, b)	Получить остаток от деления a на b

isfinite (N)	Является ли N числом
modf (N)	Определить дробную и целую часть числа N
sqrt (N)	Определить квадратный корень числа N
sin (N)	Определить синус для N-радианов
cos (N)	Определить косинус для N-радианов
tan (N)	Определить тангенс для N-радианов
degrees (N)	Перевести радианы в градусы
radians (N)	Перевести градусы в радианы

В таблице приведена только часть функций. С полным списком можно ознакомиться по [ссылке](#).

Пример:

```
from math import ceil, fabs, factorial, floor, \
    fmod, isfinite, modf, sqrt, sin, cos, tan, degrees, radians

print(f"ceil() -> {ceil(6.75)}")
print(f"fabs() -> {fabs(-4)}")
print(f"factorial() -> {factorial(5)}")
print(f"floor() -> {floor(4.34)}")
print(f"fmod() -> {fmod(9, 4)}")
print(f"isfinite() -> {isfinite(10)}")
print(f"modf() -> {modf(10.5)}")
print(f"sqrt() -> {sqrt(16)}")
print(f"sin() -> {sin(1.5708)}")
print(f"cos() -> {cos(1.5708)}")
print(f"tan() -> {tan(1.5708)}")
print(f"degrees() -> {degrees(1.5708)}")
print(f"radians() -> {radians(90)}")
```

Результат:

```
ceil() -> 7
fabs() -> 4.0
factorial() -> 120
floor() -> 4
fmod() -> 1.0
isfinite() -> True
modf() -> (0.5, 10.0)
sqrt() -> 4.0
sin() -> 0.9999999999932537
cos() -> -3.673205103346574e-06
tan() -> -272241.80840927624
degrees() -> 90.00021045914971
```

```
radians() -> 1.5707963267948966
```

Практическое задание

1. Реализовать скрипт, в котором должна быть предусмотрена функция расчёта заработной платы сотрудника. Используйте в нём формулу: (выработка в часах*ставка в час) + премия. Во время выполнения расчёта для конкретных значений необходимо запускать скрипт с параметрами.
2. Представлен список чисел. Необходимо вывести элементы исходного списка, значения которых больше предыдущего элемента.

Подсказка: элементы, удовлетворяющие условию, оформить в виде списка. Для его формирования используйте генератор.

Пример исходного списка: [300, 2, 12, 44, 1, 1, 4, 10, 7, 1, 78, 123, 55].

Результат: [12, 44, 4, 10, 78, 123].

3. Для чисел в пределах от 20 до 240 найти числа, кратные 20 или 21. Решите задание в одну строку.

Подсказка: используйте функцию **range()** и генератор.

4. Представлен список чисел. Определите элементы списка, не имеющие повторений. Сформируйте итоговый массив чисел, соответствующих требованию. Элементы выведите в порядке их следования в исходном списке. Для выполнения задания обязательно используйте генератор.

Пример исходного списка: [2, 2, 2, 7, 23, 1, 44, 44, 3, 2, 10, 7, 4, 11].

Результат: [23, 1, 3, 10, 4, 11]

5. Реализовать формирование списка, используя функцию **range()** и возможности генератора. В список должны войти чётные числа от 100 до 1000 (включая границы). Нужно получить результат вычисления произведения всех элементов списка.

Подсказка: использовать функцию **reduce()**.

6. Реализовать два небольших скрипта:
 - итератор, генерирующий целые числа, начиная с указанного;
 - итератор, повторяющий элементы некоторого списка, определённого заранее.

Подсказка: используйте функцию `count()` и `cycle()` модуля `itertools`. Обратите внимание, что создаваемый цикл не должен быть бесконечным. Предусмотрите условие его завершения.

Например, в первом задании выводим целые числа, начиная с 3. При достижении числа 10 — завершаем цикл. Вторым пунктом необходимо предусмотреть условие, при котором повторение элементов списка прекратится.

7. Реализовать генератор с помощью функции с ключевым словом `yield`, создающим очередное значение. При вызове функции должен создаваться объект-генератор. Функция вызывается следующим образом: `for el in fact(n)`. Она отвечает за получение факториала числа. В цикле нужно выводить только первые n чисел, начиная с $1!$ и до $n!$.

Подсказка: факториал числа n — произведение чисел от 1 до n . Например, факториал четырёх $4! = 1 * 2 * 3 * 4 = 24$.

Дополнительные материалы

1. [Функция `range\(\)` в Python](#).
2. [Генератор псевдослучайных чисел](#).
3. [Модуль `shutil`](#).
4. [Модуль `functools`](#).
5. [Генераторы в Python](#).
6. [Итераторы в Python](#).

Используемая литература

Для подготовки методического пособия были использованы следующие ресурсы:

1. [Язык программирования Python 3 для начинающих и чайников](#).
2. [Программирование в Python](#).
3. [Учим Python качественно \(habr\)](#).
4. [Самоучитель по Python](#).
5. [Лутц М. Изучаем Python. — М.: Символ-Плюс. 2011 \(4-е издание\)](#).