Программирование на Python. Основы

Переменные и условный оператор

Детальное изучение организации памяти и переменных в языке программирования (точнее при интерпретировании или компилировании программного кода) очень важный момент для тех, кто хочет научиться пользоваться отдельным языком на высоком уровне.

Поэтому в этой главе мы разберемся с тем, как в *Python* устроена модель хранения данных и какие простые типы поддерживаются, изучим базовые операции работы с числами и первые управляющие конструкции.

Далее рассматривается реализация *CPython*, считающаяся эталонной реализацией. Также есть и другие: *Jython*, *IronPython*, *PyPy*, *Stackless Python*.

Типы данных

Перед тем, как данные обрабатывать с помощью программы хорошо бы выяснить, с какими типами данных *Python* вообще умеет работать.

И базовых типов данных оказывается достаточно много:

Название типа	Что хранит
None	Неопределенное значение
bool	Логическое значение (истина/ложь) – *True/False*
int	Целое число
float	Вещественное число
complex	Комплексное число
list	Список значений
tuple	Кортеж (аналогия: неизменяемый список)
range	Диапазон
str	Строка
bytes	Неизменяемый набор байт (аналогия: кортеж байт)
bytearray	Набор байт (аналогия: список байт)
memoryview	Тип для работы с буфером обмена
set	Множество
frozenset	Неизменяемое множество
dict	Словарь

Типы данных, такие как *complex*, *bytes*, *bytearray* и *memoryview*, в данном учебнике рассматривать не будем ввиду экзотичности их использования. Это не значит, что они не нужны. Учебник по базовым понятиям и структурам, эти

же типы данных предназначены для решения определенных классов задач, которые в рамках данного учебника рассматриваться не будут.

Типы данных можно условно разделить на изменяемые и неизменяемые. Понимание к какому классу принадлежит тот или иной тип очень важно при работе с изменяющимися значениями.

Неизменяемые типы данных
bool
int
float
tuple
str
frozenset
range

Изменяемые типы данных
list
set
dict

Ничего странного в том, что привычные типы являются неизменяемыми. Дело в том, что такие объекты при попытке изменения значения не изменяются, вместо этого создается новый объект с новым значением.

Представление данных в памяти

Любые значения – сохраняемые для дальнейших действий или нет – перед использованием необходимо добавить в память. Перед этим *Python* каждое значение преобразует в объект *PyObject*. Чтобы не погружаться в дебри организации структуры данных в памяти, будем рассматривать *PyObject* как совокупность трех значений– тип данных (*type*), значение (*value*), количество ссылок на объект (*refcount*). Последнее значение указывает на количество ссылающихся на объект переменных.

Даже если мы хотим сделать что-то очень простое, происходит именно так. Например, чтобы вывести на экран сообщение *«Hello, world!»* запишем следующий код. Для его выполнения *CPython* сначала создаст объект с типом *str*, значением *«Hello, world!»* и количеством ссылок равным 0.

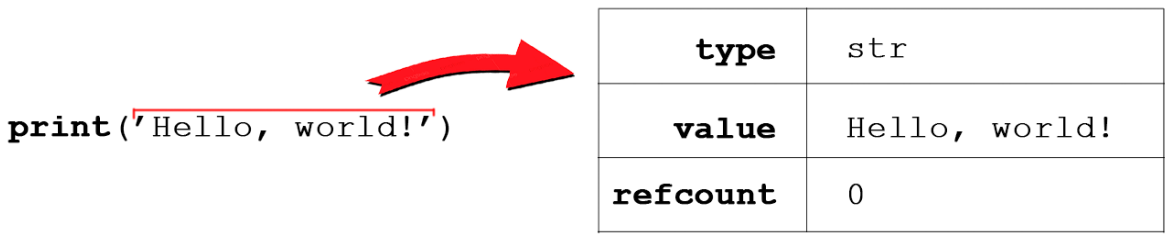


Рисунок 1. Пример объекта как *PyObject*

Все операторы и функции работают именно с представлениями в виде *PyObject*, поэтому очень полезно держать этот момент в голове.

Переменные и оператор присваивания

Исторически сложилось, что программа на любом языке программирования описывает последовательность вычислений. Причем эта последовательность может выполняться исходя из весьма сложной логики, которая описывается с помощью управляющих конструкций и структур языка программирования.

Это может быть как хрестоматийная программа для вычисления корней квадратного уравнения, так и программа для управления автопилотом автомобиля. В любой из них производится последовательность вычислений, приводящая к результату – корни уравнения или поворот колес и изменение скорости.

Да, иногда вычисления происходят не всегда в понятной человеку последовательности, однако вычисления остаются таковыми в любом случае.

При описании сложных последовательностей вычислительных преобразований возникает необходимость сохранять промежуточные вычисления. Для этого используют *переменные*.

Переменная представляет собой последовательность символов – имя – по которой можно обратиться к значению, хранящемуся в памяти.

Имя переменной составляется по следующим правилам:

- первый символ является буквой или знаком нижнего подчеркивания,

- второй и последующий символы – буквы, цифры или знаки подчеркивания.

Принято использовать «змеиный стиль» [1], то есть составные имена представлять как набор слов, разделенных знаком подчеркивания, например, *long_long_name.*

Оператор присваивания

Для определения объекта, на который будет ссылаться переменная, используется оператор присваивания.

```
имя_переменной = выражение
```

Для этого необходимо слева от оператора присваивания определить имя переменной, значение которой мы определяем, справа – записать выражение, которое вычисляет сохраняемое значение.

Также в *Python 3.8* был добавлен оператор присваивания, возвращающий значение, так называемый «моржовый оператор» (*walrus operator*).

```
имя_переменной := выражение
```

Так стало возможно записывать некоторые последовательности вычислений компактнее и использовать значения, вычисленные при проверке условий, без повторного вычисления.

Пример.

```
x = (y := 10)**2
```

В результате работы такой конструкции переменная *y* будет ссылаться на значение 10, переменная *x* – на 100.

ВАЖНО. В большинстве случаев выражение, содержащее моржовый оператор, необходимо записывать в скобках.

Так, следующая запись будет синтаксически неверна:

```
y := 10
```

В то время, как запись ниже будет правильной альтернативой варианту выше.

```
(y := 10)
```

Объект в памяти

Как же выглядит переменная в памяти? Что она хранит? И насколько вообще корректно в *Python* называть переменную переменной?

Последний вопрос больше философский, однако с позиции терминологии очень важный. Разберемся с ним в конце данного параграфа.

Простой пример (числа слева – номера строк).

```
1.  x = 500
2.  y = 300
3.  x = x - 200
```

Что же происходит в памяти?

В первую очередь определяются типы объектов в выражении справа и вычисляется значение. Для первых двух строк получится, что сначала инициализируется объект (целое число 5 для *x*, целое число 3 для *y*). Затем имя переменной сопоставится с адресом объекта.

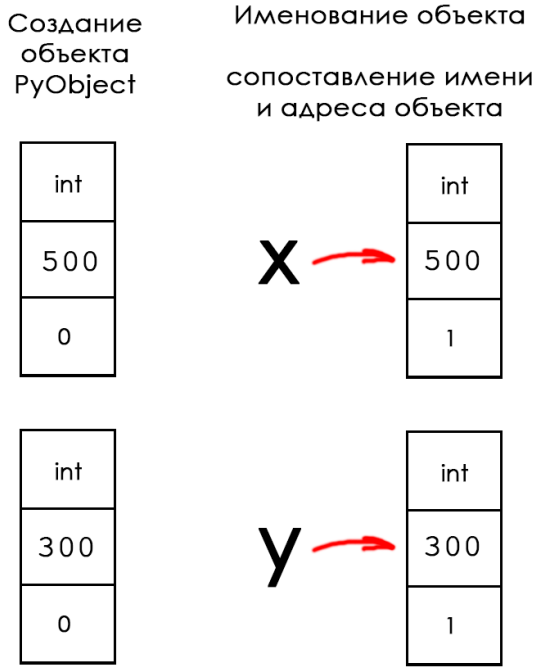


Рисунок 2. Присваивание числовых значений переменным

На этом моменте очень важно вспомнить, что *int* – неизменяемый тип данных, то есть при изменении значения в ходе вычислений мы получаем ссылку на другой объект с новым значением. Исследуем данный нюанс на примере строки 3 нашей программы.

Первый этап – вычисление выражения.

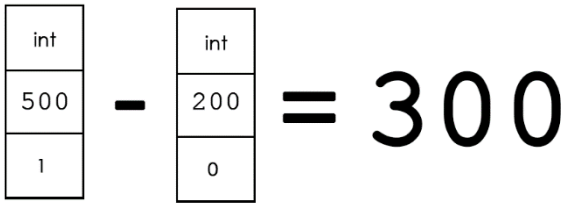


Рисунок 3. Вычисление выражения x – 200

Так как уже есть неизменяемый объект с таким значением – объект соответствующий *y* – для *x* создается соответствие с этим же объектом.

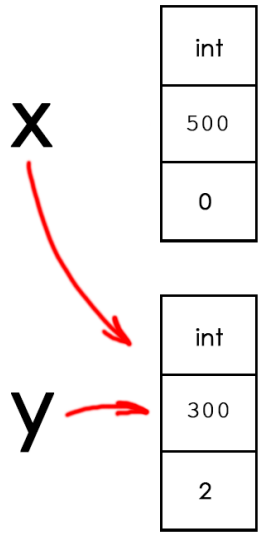


Рисунок 4. Сопоставление имени x объекту для значения 3

Также можно заметить, что для объекта числа 500 количество ссылок стало равно 0, для числа 300 – 2. Поэтому объект для числа 500 будет удален, чтобы не расходовать память на неиспользуемое значение.

Эта операция связана с процедурой оптимизации используемой памяти. В рамках данной главы мы не будем подробно рассматривать такие алгоритмы. Подробнее с данным материалом можно ознакомиться в источнике [2].

Так что же такое переменная? И переменная ли?

На самом деле в *Python* термин «переменная» скорее пришел из других языков программирования. И в *Python* переменной называется имя связанное с объектом. Это можно наблюдать в разобранном выше примере, где существует две переменные, которые ссылаются на один и тот же объект в памяти!

В реализациях таких языков программирования, как *Pascal* или *C*, для каждой переменной создается отдельный объект, который не изменяет (без нашего вмешательства) своего адреса и содержит необходимые данные (даже если они одинаковые для разных переменных). В *Python* же мы имеем дело с именованием объектов, которые существуют в памяти независимо от «переменных».

Преобразование типов

Еще одной интересной особенностью *Python* является динамическая типизация – такой принцип определения типа объекта, когда тип определяется исходя из результата выражения и используемых в нем значений.

```
s = '2134'      # str
s = 1234   # int
s = 1234.4321 # float
s = True   # bool
```

Здесь же стоит отметить, что язык *Python* является строгим динамически типизированным языком. Это означает, что если в выражении мы используем неявно определенные операции, например сложение строки и числа, то вычисление такого выражения вернет ошибку.

```
s = '2134' + '123'   # str + str = str
s = 1234 + 12        # int + int = int
s = 1234.4321 + 12.21 # float + float = float
s = 1234.4321 + 12   # float + int = float
s = True + True      # bool + bool = int 😊
s = '123' + 12       # error
```

Если необходимо преобразовать значение переменной из одного типа в другой используют функции для преобразования типов.

Функция	**Преобразуемый тип**	**Комментарий**
bool	Любой	Нулевое значение – *False* Ненулевое – *True* *bool(0) == False*
float	*int, float, complex, str*	Строка преобразуется только в случае соответствия формату числа
int	*int, float, complex, str*	При преобразовании строки можно указать систему счисления. Строка будет преобразована только если соответствует числовому представлению. Для числовых типов функция *int* выделяет целую часть.
str	Любой	

Теперь мы можем переписать ошибочное выражение, определив последовательность преобразований.

```
s = '123' + str(12)  # '12312' str + str = str
s = int('123') + 12  # 135       int + int = int
```

Перевод строки в целое число

По умолчанию преобразование строкового значения в целое число происходит в десятичную систему счисления.

```
>>> int('10232')
10232
```

Однако, можно преобразовывать строковые значения чисел, которые представлены и в других системах счисления. Для этого необходимо в качестве второго аргумента указать основание системы счисления – *int(s, b)*.

Здесь *s* – строковое представление числа в *b*-ричной системе счисления. Цифры после 10 – символы английского алфавита *a-z* в любом регистре. Поддерживается перевод чисел из систем счисления с основаниями от 2 до 36.

Пример:

```
x = int('1A20', 13)   # x = 1A20(13)= 3913(10)
```

Также существует возможность обратного перевода в системы счисления с основаниями 2, 8, 16 через форматированные строки или встроенные функции *bin, oct, hex*.

Основание	Запись (*f*-строки)	Запись (функции)
2	*f'{x:b}'*	*bin(x)*
8	*f'{x:o}'*	*oct(x)*
16	*f'{x:x}'*	*hex(x)*

Заметим, что тип результата – строка. Также отметим, что встроенные функции в начале строки возвращают префикс – *0b, 0o* и *0x* для 2, 8 и 16 систем счисления соответственно.

Примеры.

Перевести число 223 в строковое двоичное представление.

f-строка

```
>>> f'{223:b}'
'11011111'
```

Функции

```
>>> bin(223)
'0b11011111'
```

Перевести значение переменной *var (1443)* в строковое шестнадцатеричное представление.

f-строка

```
>>> f'{var:x}'
'5a3'
```

Функции

```
>>> hex(var)
'0x5a3'
```

ВАЖНО №1: с помощью форматирования нельзя перевести в любую систему счисления. Для этого используют специальные алгоритмы, которые мы рассмотрим в будущем.

ВАЖНО №2: при работе с *f*-строками и переводом в 2, 8, 16 системы счисления используются СИМВОЛЫ (*b, o, x*), обозначающие основание системы счисления. Подробнее о *f*-строках поговорим в дальнейших главах.

Множественное присваивание

В данном параграфе множественное присваивание рассмотрим в более узком понятии, чем оно есть. Другие особенности данного механизма будем вводить постепенно в последующих главах.

На данном этапе для нас важно, как минимизировать код или сделать его более читаемым.

Python синтактически поддерживает возможность задавать значения нескольких переменных с помощью одного оператора присваивания.

Для этого слева от оператора присваивания надо указать все имена переменных, которые мы хотим изменить/инициализировать, справа – соответствующее количеству переменных слева количество значений.

```
a, b = 1000, 450 + 550
```

Интересный факт!

При запуске такого кода в оболочке *REPL* оба имени *a* и *b* будут ссылать на один объект.

Также существует альтернативный способ задать для нескольких переменных один и тот же объект.

```
a = b = 1000
```

В таком случае создается один объект со значением 1000 и адрес объекта записывается в переменные *a* и *b*.

Операции

Для обработки и преобразования данных нужно определиться, какие операции есть в нашем распоряжении. В данном параграфе рассмотрим базовые арифметические и логические операции, а также приоритет их выполнения.

Арифметические операции

Результатом арифметической операции является число. Надо отметить, что в арифметическом выражении могут участвовать и данные логического типа. В таком случае *True* интерпретируется как 1, *False* – как 0.

Операция	**Обозначение**	**Пример использования**
Унарный минус	-	-10
Сложение	+	`2 + 2     # 4` `False + 3 # 3`
Вычитание	–	`10 - 18  # -8` `True - 5 # -4`
Умножение	*	`4*5     # 20`
Деление	/	`10 / 4  # 2.5`
Возведение в степень	**	`2**4    # 16`

Целочисленное деление	//	`10 // 2   # 5`
		`13 // 5   # 2`
		`-12 // 7   # -2`
		`13 // -6  # -3`
		`-10 // -3  # 3`
		`10 // 2.6 # 3.0`

Остаток от деления	%	`10 % 2   # 0`
		`13 % 5   # 3`
		`-12 % 7   # 2`
		`13 % -6  # -5`
		`-10 % -3  # -1`
		`10 % 2.6 # 2.2`

Взятие модуля Абсолютное значение	abs(x)	`abs(-10)  # 10` `abs(1000) # 1000`

Операция «унарный минус» делает из числа (подвыражения) в начале выражения отрицательное значение. Так при записи -10 + 20 сначала выполнится операция унарного минуса «-10» и только затем произойдет сложение «-10» и «20».

Функции целочисленного деления и нахождения остатка от деления работают по следующему правилу: если число a не делится нацело на b, то знак остатка должен совпадать со знаком делителя. Или соблюдается одно из следующих условий.

$$\begin{bmatrix} 0 \le r < b \\ b < r \le 0 \end{bmatrix}$$, где b – делитель, r – остаток.

Результат целочисленного деления в таком случае можно представить, как

$a//b = \left\lfloor \frac{a}{b} \right\rfloor$, где $\lfloor A \rfloor -$ округление вниз (до меньшего или равного целого).

Соответственно, $a\%b = a - (a//b) \cdot b$

Поэтому

$$-12//7 = \left\lfloor \frac{-12}{7} \right\rfloor = \left\lfloor -1\frac{5}{7} \right\rfloor = -2$$

$$-12\%7 = -12 - (-12//7) \cdot 7 = -12 - (-2) \cdot 7 = -12 + 2 \cdot 7 = 2$$

Или

$$13//(-6) = \left\lfloor \frac{13}{-6} \right\rfloor = \left\lfloor -2\frac{1}{6} \right\rfloor = -3$$

$$13\%(-6) = 13 - 13\%(-6) \cdot 7 = 13 - (-3) \cdot (-6) = 13 - 18 = -5$$

Также данные операции могут быть применены к вещественным числам.

Логика вычисления остается такая же – результат целочисленного деления является вещественным числом с нулевой дробной частью, результат для остатка от деления вычисляется по приведенной выше формуле.

Существует и другой подход к работе с остатками, когда остаток не может быть отрицательным. Например, остаток от деления 13 на -6 будет равен 1, а сам результат целочисленного деления равен 2.

При этом формула

$$a = b \cdot r + q$$

где b – делитель, r – результат целочисленного деления, q – остаток, справедлива для обоих случаев (*Python* и данного определения).

Знаки сравнения

Для числовых данных.

Операция	**Обозначение**	**Пример использования**
Равно	==	`4+2 == 3+3  # True` `False == True # False`
Не равно	!=	`3+1 != 2+2 # False`
Больше	>	`2 > 2 - 3  # True`
Больше или равно	>=	`2 >= 1+1  # True`
Меньше	<	`3 < 10  # True`
Меньше или равно	<=	`5 <= 1  # False` `True <= False # False`

При использовании операторов присваивания с логическими значениями, последние сначала преобразуются в числовое представление – *True* в 1, *False* в 0.

Посмотрим, какие значения получаются в таком случае.

A	*B*	==	!=	>	>=	<	<=
False	*False*	1	0	0	1	0	1
False	*True*	0	1	0	0	1	1
True	*False*	0	1	1	1	0	0
True	*True*	1	0	0	1	0	1

Также в *Python* есть механизм цепочек сравнения (*chaining comparison operators*) с помощью которого, в числе прочего, удобно обозначать диапазоны значений числовых переменных.

Пример.

Число x в диапазоне [10; 110].

```
10 <= x <= 110
```

Про сравнение объектов других типов будет рассказано в следующих главах.

Логические операции

Логические операции применяются для связки логических выражений.

Операция	**Обозначение**	**Пример использования**
Отрицание	`not`	`not True  # False`
Дизъюнкция	`or`	`False or True  # True`
Конъюнкция	`and`	`True and True  # True`

В языке *Python* всего три логические операции. В качестве альтернативы можно использовать, например, оператор <= для операции импликации и оператор == для эквивалентности. Однако стоит помнить о порядке выполнения операций в записываемом выражении.

Примеры.

Число x одновременно четно и больше 10.

```
x % 2 == 0 and x > 10
```

Число оканчивается на 5 или принадлежит промежутку $(-\infty;0)U(100;+\infty)$

```
x % 10 == 5 or 0 >= x <= 100
```

Любое выражение в *Python* вычисляется слева направо, поэтому при вычислении значения логического выражения выражение может быть вычислено досрочно. Выражение считается вычисленным, если результат левого операнда дизъюнкции равен *True* или результат левого операнда конъюнкции равен *False*. Так как нет смысла вычислять значение выражения `1 or expr` или `0 and expr`, потому что результат первого всегда будет истинным, второго – ложным.

Пример вычисления с «досрочным» результатом:

```
a = 10
b = 15
c = 20
#  True  or  ...  == True
if a < b or c > a + b:
  c = 10
```

На этом принципе построены некоторые алгоритмы проверок значений. Например, при работе с конъюнкцией правильнее писать в качестве левой части выражения самое редко встречающееся условие, тогда записанные правее условия будут проверяться только при истинности самого редко срабатывающего условия.

Побитовые операции

Применяются для обработки целых чисел в двоичном представлении.

Операция	**Обозначение**	**Пример использования**
Побитовое отрицания	`~a`	`~5 # -6`
Побитовое умножение	`a & b`	`13 & 10 # 8`
Побитовое сложение	`a | b`	`13 | 6 # 15`
Побитовое исключающее или	`a ^ b`	`13 ^ 6 # 11`

Побитовое отрицание производит инвертирование бит, в том числе бита, отвечающего за знак [3]. Поэтому в результате получается значение *–(a+1)*.

Сокращенная запись

Для действий, направленных на изменение значения переменной относительно текущего значения могут применяться сокращенные операторы.

Операция	**Обозначение**	**Эквивалент**
Сложение	`a += e`	`a = a + e`
Вычитание	`a -= e`	`a = a - e`
Умножение	`a *= e`	`a = a * e`
Деление	`a /= e`	`a = a / e`
Целочисленное деление	`a //= e`	`a = a // e`
Остаток от деления	`a %= e`	`a = a % e`
Побитовое умножение	`a &= e`	`a = a & e`
Побитовое сложение	`a |= e`	`a = a | e`
Побитовое исключающее или	`a ^= e`	`a = a ^ e`

Сокращенные операторы выполняются всегда в самом конце, после вычисления выражения справа от оператора.

Приоритет выполнения операций

В любом выражении можно задавать порядок выполнения операций с помощью скобок. В остальных случаях нужно знать, в каком порядке выполняются операции в выражении.

Приоритет выполнения операций следующий

Операция
`**`
`~`
`*, /, //, %`
`+, -`
`&`
`|`
`^`
`==, !=, <, <=, >, >=`
`not`
`and`
`or`

Говоря иначе, чем выше операция в таблице, тем раньше она выполняется. Если несколько операций записаны в одной строке, то они имеют одинаковый приоритет. При наличии их в одном выражении, выполняются они слева направо, то есть в первую очередь выполняется самая левая одноприоритетная операция.

Пример (рис.4-5)

```
  1   3    2   4       9    7   5    6   8
a ** 4 + b * c >= 100 and 5 ^ c / 10 * 2 <= 50
```

Рисунок 5. Порядок выполнения операций в выражении

```
(((a ** 4) + (b * c)) >= 100) and ((5 ^ ((c / 10) * 2)) <= 50)
```

Рисунок 6. Расстановка приоритета операций с помощью скобок

Стоит заметить, что почти все операции являются лево ассоциативными, то есть при совпадении приоритета у соседних операций сначала выполняется операция, которая левее, затем правая.

Единственное исключение – операция возведения в степень, являющаяся право ассоциативной.

Например, запись $a**b**c$ будет вычисляться, как a^{b^c}, а не как $(a^b)^c$.

Ввод и вывод данных

Для коммуникации с программным кодом при запуске программы, в программу вставляют специализированные функции – ввода и вывода.

Чтобы программа обработала какое-либо значение, его необходимо ввести. Сделать это можно по-разному – считать из файла, запросить у сервиса в интернете или просто ввести через терминал.

Разберем самый простой случай.

Ввод значения через терминал.

Для этого в *Python* есть функция *input()*.

Функция *input()* возвращает поступившее через терминал сообщение до переноса строки.

ВАЖНО: результатом работы функции *input()* является строковое значение.

Поэтому если нужно ввести число, то считанное строковое значение необходимо преобразовать с помощью соответствующей функции преобразования – *int()* или *float()*.

Пример – ввод целого числа из терминала.

```
# ввод строки из терминала
inp = input()
# преобразование введенной строки в целевое значение
x = int(inp)
```

Также данную операцию можно выполнить в одну строку, передав результат выполнения функции input() на вход функции для преобразования.

```
x = int(input())
```

Для интерактивности можно в качестве аргумента функции input привести приветственную строку. Например, *«Input x: »*.

```
x = int(input('Input x:'))
```

Вывод значений на экран

Для вывода сообщений на экран используется функция *print()*.

Функция принимает на вход значения или имена переменных через запятую, которые необходимо вывести. Строго говоря, данная функция имеет ряд настроек, которые нужны для корректировки формата вывода. Однако на данном этапе для нас хватит и уже описанного функционала.

Пример – вывод через пробел значений переменных *a* и *b*.

```
print(a, b)
```

Пример простой программы для нахождения корня линейного уравнения

$kx+b = 0$.

```
# ввод коэффициента k
k = float(input('Input k:'))
# ввод коэффициента b
b = float(input('Input b:'))
# вычисление корня
x = -b / k
# вывод корня на экран
print('x =', x)
```

Условный оператор

Условный оператор – конструкция, управляющая цепочкой вычислений, которая позволяет выбрать ОДИН ИЗ *n* вариантов дальнейшего выполнения алгоритма.

Говоря на языке примеров, у нас есть *n* возможных вариантов вычислений, чтобы для каждого из них реализовать различные алгоритмы мы применяем конструкцию «если *вариант1*, то *делаем1*, иначе если *вариант2*, *делаем2*, …, иначе если *вариантN* то *делаемN*, во всех не перечисленных случаях *делаем(N+1)*».

Условный оператор является первым из рассматриваемых в этом учебнике составным оператором. Составным он называется, потому что состоит из нескольких частей – ключевых слов (*if*, *elif* *else*) и блоков команд, выполняемых при истинности одного из условия (про блоки команд подробнее ниже).

Подробную документацию по этому и другим операторам можно найти в официальной документации [4] или в неофициальном русском переводе [5].

Условный оператор можно представить, как

```
условный_оператор ::=  "if" условие ":" блок команд
                     ("elif" условие ":" блок команд)*
                     ["else" ":" блок команд]
```

В представленной нотации запись (…)* означает, что на этом месте может быть 0 или больше выражений, описанных в скобках, […] – может быть 0 или 1 выражение описанного в скобках формата.

Говоря иначе, допустимы, например, такие использования

```
условный_оператор ::=  "if" условие1 ":" блок команд 1
```

Или

```
Условный_оператор ::=  "if" условие1 ":" блок команд 1
                       "elif" условие2 ":" блок команд 2
                       "elif" условие3 ":" блок команд 3
```

Или

```
условный_оператор ::=  "if" условие1 ":" блок команд 1
                       "elif" условие2 ":" блок команд 2
                       "else" ":" блок команд 3
```

Или

```
условный_оператор ::=  "if" условие1 ":" блок команд
                       "else" ":" блок команд 2
```

Самая важная особенность – условный оператор выбирает не более одного блока команд, который будет выполнен. В случае, если ни одно условие не выполняется и не задана секция *else*, условный оператор не перейдет ни к какому блоку команд и передаст управление команде, следующей сразу после последнего блока команд, описанного в условном операторе.

ВАЖНО. Условия проверяются в той последовательности, в которой они указаны в последовательности блоков *elif*.

Например, следующие обобщенные алгоритмы не являются эквивалентными:

```
Условный_оператор ::=  "if" условие1 ":" блок команд 1
                       "elif" условие2 ":" блок команд 2
                       "elif" условие3 ":" блок команд 3
```

```
Условный_оператор ::=  "if" условие1 ":" блок команд 1
                       "elif" условие3 ":" блок команд 3
                       "elif" условие2 ":" блок команд 2
```

Если мы нумеруем условия в порядке их указания в составном условном операторе, то логика проверки условий и, соответственно, выбора блока команд будет следующая:

> Если выполняется условие1, то выполняем блок команд 1 и завершаем работу условного оператора, иначе, если выполняется условие 2, то выполняем блок команд 2 и завершаем работу условного оператора, …, если не выполняется ни одно условия, выполняем блок команд, указанный после else (если блок указан) либо завершаем работу условного оператора (если блок команд для else не указан).

Данный процесс можно представить в виде блок схемы (рис.7).

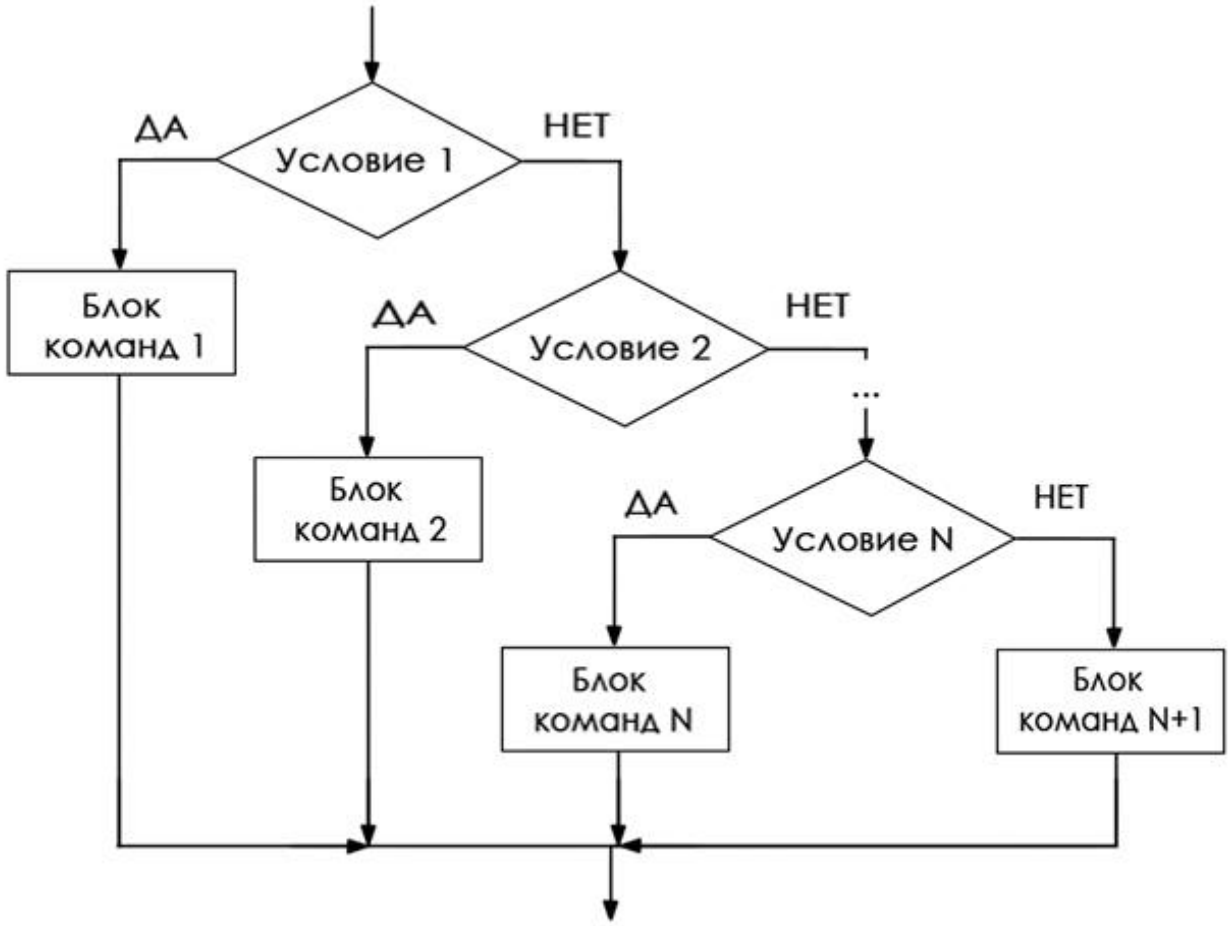


Рисунок 7. Схематическое изображение работы условного оператора

В случае отсутствия секции блока *else* программа не выполнит ничего во время работы условного оператора, когда ни одно из условий не будет истинным (блок команд *N+1* будет отсутствовать). Если в условном операторе не будет блоков *elif*, то условный оператор выполнит блок команд 1, если условие выполняется, и блок команд *N+1*, если не выполняется. Когда же нет ни блоков *elif*, ни блока *else*, алгоритм выполнит блок команд 1, если условие истинно, и не выполнит ничего, если условие ложно.

Блоки команд

В языке *Python* доступно два варианта оформления блоков команд в составных операторах, в том числе и условном операторе.

1) На той же строке, что и ключевые слова – *if*, *elif* и *else* для условного оператора – через точку с запятой.

Пример:

```
if x % 2 == 0: y = y +x; z = 10
else: y = 0; z = 13
```

2) Каждая команда с новой строки и уровнем отступа на 1 больше, чем у ключевых слов *if*, *elif* и *else*

Тот же пример:

```
if x % 2 == 0:
  y = y +x
  z = 10
else:
  y = 0
  z = 13
```

Второй способ считается общепринятым и соответствует официальным рекомендациям по оформлению кода *PEP-8* [1].

В чем же заключается идея такого формата.

Все блоки команд в составном операторе являются вложенными. Чтобы показать уровень вложенности для блока команд в таком операторе увеличивают отступ относительно ключевого слова. Также очень важно, чтобы все дальнейшие составные команды использовали аналогичное количество пробелов в качестве отступа.

Чтобы не считать каждый раз пробелы принято устанавливать отступы с помощью знака табуляции (кнопка *Tab* на клавиатуре, обычно, над *Caps Lock*). Команды одного уровня вложенности в рамках одного составного оператора должны иметь одинаковый отступ от начала строки.

Строго говоря, *Python* не запрещает делать разные отступы для обособленных составных операторов. Однако, такой подход противоречит рекомендациям *PEP-8* и усложняет поддержку написанного кода.

Для иллюстрации приведем два примера – с разными отступами и с одинаковыми. Данные примеры наглядно показывают удобство применения рекомендации *PEP-8* для оформления блоков команд.

Пример 1 (разные отступы)

```
if x % 2 == 0:
    y = y + x
    if y > 5:
     y = 6
if x // 3 == 4:
           y = 0
           if x > 10:
                 z = 22
```

Пример 2 (одинаковые отступы)

```
if x % 2 == 0:
    y = y + x
    if y > 5:
          y = 6
if x // 3 == 4:
     y = 0
     if x > 10:
          z = 22
```

Тернарная условная операция

Для удобства в некоторых случаях используют тернарную условную операцию. Обычно, она используется, когда в зависимости от условия устанавливает значение конкретной переменной.

Пример (в формате обычного условного оператора):

```
if y > 10:
     x = 5
else:
     x = y + 3
```

Пример (с помощью тернарной условной операции):

```
x = 5 if y > 10 else y + 3
```

Тернарную условную операцию удобно применять, когда условие небольшое и выражение, которое присваивается, в обоих случаях получается по простому алгоритму. Иначе анализ тернарной условной операции становится гораздо сложнее и увеличивает сложность поддержки кода. В качестве маркера «понятного» выражения можно использовать, например, рекомендации Дзен *Python* [6] и не делать строки длиной более 79 символов.

Запуск и трассировка программы

Мы подходим к практике. Точнее пока что больше к попыткам запустить написанную программу и исследовать, как она работает. Для этого нам нужен, собственно, сам интерпретатор и самый простой инструмент для написания кода – *IDLE* [7].

После скачивания установочного файла по ссылке и его установки у нас появляется доступ к *IDLE* (обычно это ярлык в меню «Пуск»). Открываем его и компьютер готов к экспериментам.

После запуска появляется окно Python Shell, здесь ты можешь покомандно вводить программу и смотреть, что получается. В рамках данного параграфа нас интересует возможность запускать написанные программы (скрипты).

Для этого нужно наш код оформить в виде файла.

Алгоритм действий:

1) *File -> New file* (или *Ctrl+N*)
2) Пишем код, который хотим запустить
3) Сохраняем написанный код в файл (важно делать перед каждым запуском)
4) Запускаем (*Run -> Run module* или *F5*)

Теперь мы умеем создать файл с программой и запустить его. После запуска мы можем проанализировать вывод нашей программы. Например, результат может быть как на рис.8.

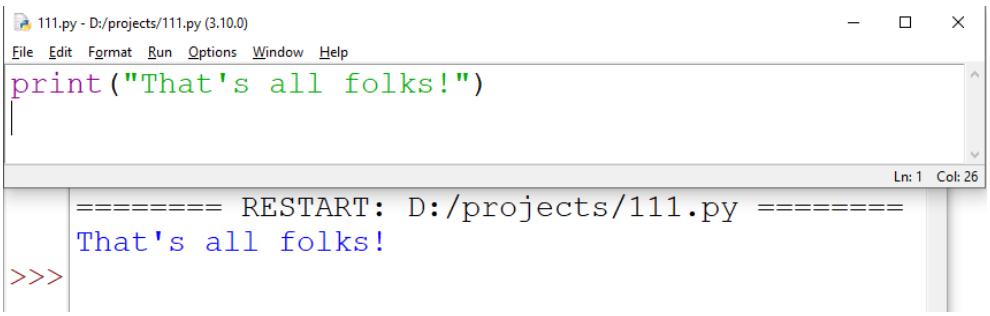


Рисунок 8. Запуск простой программы и результат

Инструмент Debug

Однако при разработке нашего алгоритма мы можем допустить логические ошибки в коде, в связи с чем наш код будет делать не то, что мы хотим.

Отследить поведение программы с помощью вывода промежуточных вычислений с помощью функции *print*, конечно, можно. Но такой метод может оказаться весьма неудобным, если мы имеем дело, например, с многострочной программой или циклом с большим количеством повторов.

Неудобный он в первую очередь из-за возможно длинного вывода и неудобства анализировать таковой.

Поэтому для пошагового анализа работы программы используют специализированные средства отладки – отладчики, также именуемые дебагерами (от *de* – отрицание действия, *bug* – программная ошибка [8]).

Для запуска *Debug*-режима в IDLE необходимо выбрать в меню *Debug -> Debugger*. Теперь при запуске скрипта код будет выполняться по одной инструкции при нажатии на кнопку *Step*, и по одной строке при нажатии на кнопку *Over*.

Режим *Step* выполняет все инструкции внутри вызываемых методов, поэтому анализ хода выполнения программы получается существенно длиннее. *Over* же работает построчно и позволяет в ускоренном режиме проанализировать работу написанного кода.

В нижней области окна отладчика можно контролировать значения переменных на текущем этапе

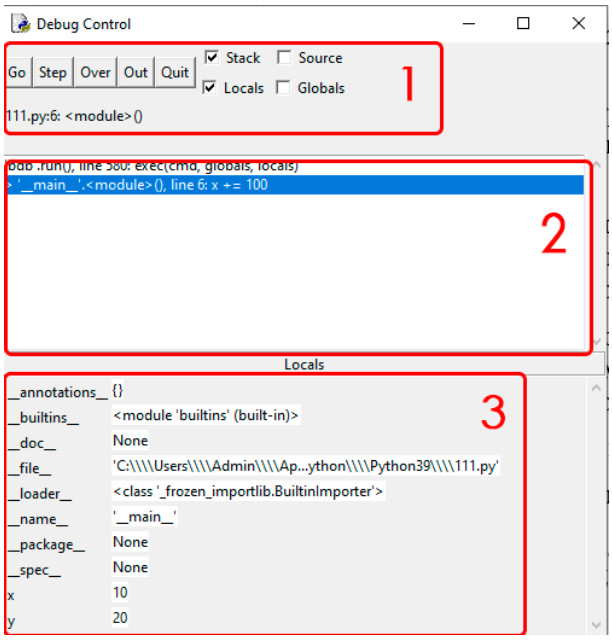


Рисунок 9. Окно дебагера IDLE

Область 1 – панель управление ходом выполнения программы.

Go – продолжить выполнение до конца,

Step – выполнить следующую команду,

Over – выполнить текущую строку,

Out – выполнить текущую функцию до конца (предполагает, что сейчас отладчик находится внутри функции),

Quit – закончить работу отладчика.

Про функции мы поговорим в одной из следующих глав. На текущем этапе для нас наиболее полезные кнопки – *Go* и *Over*.

Область 2 – стек вызовов [9].

Область 3 – состояние памяти.

Здесь отображаются значения служебных переменных и тех, которые мы используем в ходе выполнения программы.

Для удобства анализа полезно поставить галочку на пункте *Source*, тогда в соседнем окне будет подсвечиваться текущая выполняемая строка (рис.10).

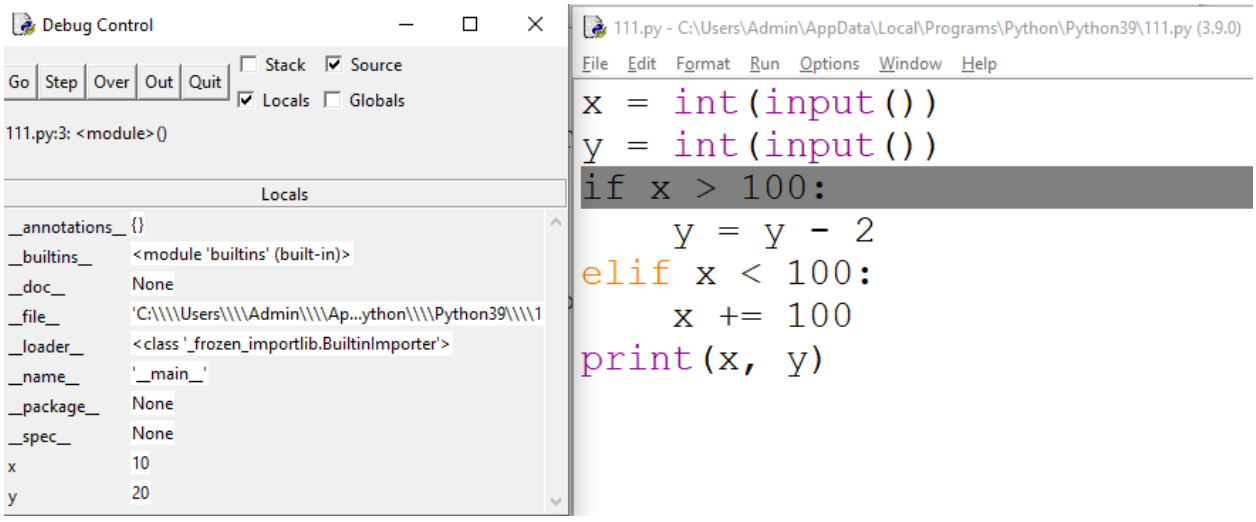


Рисунок 10. Пример отображения окна отладчика и исходного кода

Про объекты в памяти

Если начать писать программу в оболочке *REPL* для *Python* (*REPL – read, eval, print, loop*), можно заметить одну особенность. Значения в диапазоне [-5; 256] будут иметь всегда один и тот же адрес для разных переменных, другие значения при их именовании будут иметь разные адреса для разных имен в случае совпадения значений.

Так, при попытке выполнить следующую последовательность команд,

```
1. >>> a = 100
2. >>> b = 55 + 45
```

имена *a* и *b* будут ссылаться на один и тот же объект.

В то время, как при следующей последовательности вычислений

```
1. >>> a = 1000
2. >>> b = 550 + 450
```

имена *a* и *b* будут ссылаться на разные объекты.

REPL	*Скрипт*

`>>> a = 100`
`>>> b = 55 + 45`
`>>> id(a), id(b)`
`(1342269254224, 1342269254224)`
 	
`>>> a = 1000`
`>>> b = 550 + 450`
`>>> id(a), id(b)`
`(1342341491280, 1342341490800)`
	`a = 100`
	`b = 55 + 45`
	`print(id(a), id(b))`
	
	`a = 1000`
	`b = 550 + 450`
	`print(id(a), id(b))`
	
	`2037007603152 2037007603152`
	`2037013785168 2037013785168`

Функция *print* выводит переданные ей значения на экран через пробел.

Это объясняется тем, что при запуске интерпретатора *CPython* создаются объекты для значений в диапазоне [-5; 256], так как считается, что это наиболее часто используемые значения. Такой подход позволяет не тратить время на выделение памяти для них. Аналогично кэшируются при первом использовании строковые значения, состоящие из букв, цифр и символов подчеркивания длиной до 20 символов.

При запуске же скрипта целиком происходит кеширование всех используемых неизменяемых объектов. Эта процедура относится к оптимизации использования памяти и в данном разделе не рассматривается. Данный факт можно проверить с помощью функции *id(var_name)*, которая возвращает адрес объекта, на который ссылается переменная *var_name*.

Последовательности и итераторы

Последовательности

В *Python* есть так называемые типы-последовательности (*Sequences*). Последовательность – общий термин, который предполагает упорядоченный набор данных. Говоря иначе, элементы в объекте-последовательности будут расположены в том же порядке, в котором будут добавлены, либо объявлены (в зависимости от того, изменяемая последовательность или нет).

К последовательностям относят типы: *str*, *list*, *tuple*, *bytes*, *bytesarray* и *range*.

Обращение к элементам объекта последовательности происходит путем указания номера (индекса) элемента в последовательности. При этом элементы последовательности нумеруются от первого к последнему, начиная с нуля, или от последнего к первому в обратном порядке, начиная с -1.

Так, если последовательность S содержит 5 элементов, то обращение к S_3 эквивалентно обращению к S_{-2} (рис.1).

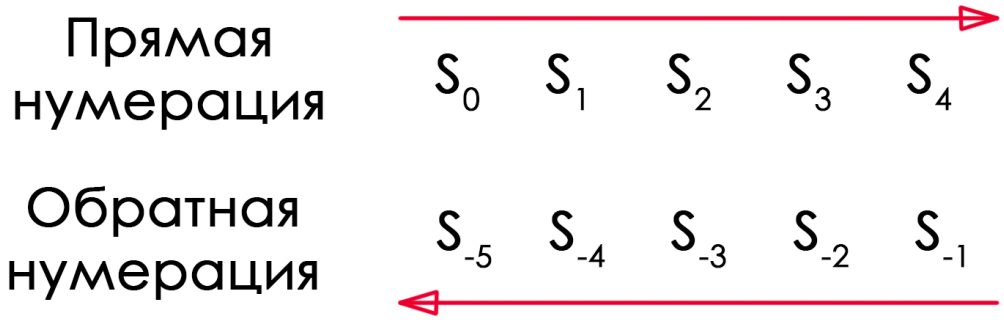


Рисунок 1. Прямая и обратная нумерация в Python

Для обращения к элементу последовательности по индексу используется синтаксическая конструкция `s[index]`, где *s* – последовательность, к элементу которой необходимо обратиться, *index* – индекс элемента в прямой или обратной нумерации.

При попытке выхода за границы последовательности (указание несуществующего индекса), интерпретатор вернет ошибку «*IndexError: <sequence> index out of range*» или «*Ошибка обращения по индексу: индекс выходит за допустимый диапазон*».

ВАЖНО: допустимые индексы элементов последовательности всегда находятся в диапазоне `[-len(seq); len(seq)-1]`, где *len*(*seq*) – количество элементов последовательности

Методы работы с каждым из перечисленных выше типов мы разберем подробнее чуть ниже. Сейчас же нам важно понимать, как обозначаются (инициализируются) значения для данных типов.

Тип	**Пример**	**Описание**
Строка (*str*)	`s = '12345abcde'`	Строка из 10 символов. `s[0]` – '1', `s[1]` – '2', …, `s[9]` – 'e'.
Кортеж (*tuple*)	`s = (1, 2, '123', 5)`	Неизменяемая последовательность из 4 объектов `s[0]` – 1, `s[1]` – 2, `s[2]` – '123', `s[3]` – 5.
Список (*list*)	`s = [5, 10, 'asd']`	Изменяемая последовательность из 3 объектов `s[0]` – 5, `s[1]` – 10, `s[2]` – 'asd'.

Стоит заметить, что для создания пустой последовательности необходимо оставить незаполненным пространство между границами последовательности:

- `s = ''` для строки,
- `t = ()` для кортежа,
- `l = []` для списка.

Также надо помнить, что кортеж из одного элемента определяется несколько неочевидным образом. Так как при отсутствии нескольких значений в круглых скобках, записанное значение интерпретируется, как отдельный объект – результат выполнения операций в скобках.

Например, при записи `x = (3)` получим объект типа *int* со значением 3. То есть такие скобки считаются скобочными группами в выражении, так же как в выражении `x = (25 - y) * 5`. Ведь если бы это было не так, то подвыражение `(25 - y)` воспринималось бы как кортеж, что привело бы к неправильному результату при вычислении данного выражения.

Поэтому для формирования кортежа из одного элемента кортеж записывают как одноэлементный кортеж (*singleton tuple*) с помощью запятой, после которой не указывается следующий элемент.

```
t = (1,)
t = 1,
```

Строго говоря, также мы можем не указывать последний элемент и при инициализации списка. Однако, обычно в этом нет необходимости.

Общие функции, операторы и методы для последовательностей

Последовательности поддерживают ряд общих методов и операторов [10].

Метод	**Обращение**	**Пример**
Количество элементов	`len(seq)`	`len('1234')  #4` `len([1,4,8])  #3`
Конкатенация (сложение)	`seq1 + seq2`	`'asd'+'123' #'asd123'` `[5]+[1,3]  #[5,1,3]`
Повторение *N* раз	`seq*N` `N*seq`	`'a'*4  # 'aaaa'` `2*[4, 2]#[4, 2, 4, 2]`
Проверка вхождения *value*	`value in seq`	`'4' in '123234'#True` `'0' in '123'  #False`
Проверка НЕвхождения *value*	`value not in seq`	`'1' not in '22'#True` `'2' not in '22'#False`
Количество вхождений *value*	`seq.count(value)`	`'12311'.count('1') #3`
Индекс первого (левого) элемента *value*	`seq.index(value)`	`'12511'.index('5') #2`
Максимальный элемент	`max(seq)`	`max([5,2,10,2])  #10` `max('12345asdfw')#'w'`
Минимальный элемент	`min(seq)`	`min([5,2,10,2])  #2` `min('12345asdfw')#'1'`
Сортировка элементов	`sorted(seq, reverse=True/False)`	`sorted('bac')` `# ['a', 'b', 'c']`

Функция *sorted* всегда возвращает список элементов последовательности, поданной в качестве аргумента.

Важно, что при операциях поиска (*count, index, in, not in*) тип элемента *value* был совместим с объектом последовательности. Например, если мы производим поиск в строке, то нельзя в качестве искомого значения использовать любые значения, кроме строк.

При вызове методы *index,* если элемент отсутствует в последовательности, интерпретатор вернет ошибку «*ValueError: <value> is not in <sequence>*» или «Ошибка значения: *значение* отсутствует в *последовательности*».

Методы *count, index* и операторы *in, not in* для строки позволяют искать не только единичные символы, но и подстроки.

Примеры

```
'15243'.index('2')      # 2
'15243'.index('43')     # 3
'12112'.count('1')      # 3
'12112'.count('12')     # 2
[1, 2, 3].index(2)      # 1
[1, 2, 3].index([2, 3]) # ValueError
[1, 2, 3].count(3)      # 1
[1, 2, 3].count([2, 3]) # 0
```

В примере `[1, 2, 3].index([2, 3])` получаем *ValueError* так как в списке нет ЭЛЕМЕНТА, содержащего список [2, 3]. В отличии от следующей записи

```
[1, 2, [3, 4]].index([3, 4]) # 2
```

Сравнение последовательностей

В общем случае последовательности сравниваются поэлементно. Сравнение допустимо только между последовательностями одного типа. При сравнении на равенство между различными типами последовательностей результатом будет ложь. При попытке сравнения порядка – ошибка несовместимости типов.

```
>>>a, b, c = ['1', '2'], ('1', '2'), '12'
>>>a == b
False
>>>b == c
False
>>>a > b
TypeError: '>' not supported between instances of 'list' and
'tuple'
```

Признак равенства – все элементы S соответствуют элементам F или $s_i = f_i$ и $f_j = s_j$ для всех допустимых i и j. Говоря иначе последовательности F и S должны быть одинаковой длины и элемент, стоящий на позиции i в F, должен быть равен элементу на i-той позиции S.

Последовательности сравниваются лексикографически – слева направо. То есть операция сравнения применяется последовательно к элементам последовательности, начиная с самого левого (нулевого) элемента. И происходит до тех пор, пока либо не дойдет до конца обоих последовательностей (в случае равных последовательностей), либо не найдет отличия в одном из элементов. Если одна из последовательностей является началом другой, то последовательность большей длины считается последовательностью с бóльшим значением.

Пример.

```
'1234' < '12057'
```

	Элемент левой последовательности	Элемент правой последовательности	Результат сравнения
Шаг 1	'1'	'1'	==
Шаг 2	'2'	'2'	==
Шаг 3	'3'	'0'	>

Первая пара отличающихся элементов последовательностей – символы '3' и '0'. Данные элементы противоречат проверяемому условию ('3' > '0'), следовательно, результат сравнения равен *False*.

Важно!

Последовательности сравнимы только в том случае, если между соответствующими элементами определены операции сравнения. Если же это не так, то интерпретатор вернет ошибку *«TypeError: <operation> not supported between instances of <type1> and <type2>»* или «Ошибка типа: <операция> не поддерживается между <тип1> и <тип2>».

Это же замечание применимо для функции сортировки – *sorted* может отсортировать только такую последовательность, все элементы которой могут быть сравнимы между собой.

Пример.

```
>>>a = [1, 2, 3]
>>>b = [1, 2, '3']
>>>a > b
TypeError: '>' not supported between instances of 'int' and
'str'
```

Операция сравнения возвращается ошибку, так как элементы *a*[2] и *b*[2] имеют разные типы, операция «>» над которыми не определена.

Срезы

Срезом называется подпоследовательность последовательности от элемента *start* до элемента *finish* с шагом *step*. Причем элемент с индексом *start* включается в срез, элемент с индексом *finish* – не включается. Результатом взятия среза является новый объект того же типа, что и последовательность из которой извлекается срез.

Значение шага *step* – целое число, не равное 0.

```
seq[start:finish:step]
```

Каждый из параметров среза является необязательным.

- при неуказанном параметре *start* – значение считается 0 при положительном *step* и -1 при отрицательном;
- при неуказанном параметре *finish* – последовательность обрабатывается до конца при положительном *step*, и до начала при отрицательном;
- значение *step* по умолчанию равно 1.

Срез будет непустым, при:

- положительном шаге и положении индекса *start* левее индекса *finish,*
- отрицательном шаге и положении индекса *start* правее индекса *finish.*

Во всех остальных случаях срез будет пустым.

Примеры (помним, что элементы нумеруются с нуля).

```
'abcdefgh'[2:6:3] == 'cf'      # 2 и 5 элементы
'abcdefgh'[5:1:-1] == 'fedc'   # 5, 4, 3, 2 элементы
'abcdefgh'[-4:6] == 'ef'       # 5, 6 или -4, -5
'abcdefgh'[::-1] == 'hgfedcba'# задом-наперед
```

Можно заметить, что при указании индексов в разном направлении нумерации в рамках одного среза срез берется от указанного слева ЭЛЕМЕНТА до указанного справа ЭЛЕМЕНТА и не продолжается в обратном направлении.

Например, срез `'012345678'[-5:8:2]` не будет перебирать значения индексов в диапазоне [-5; 8] с шагом 2 – [-5, -3, -1, 1, 3, 5, 7] . Вместо этого будет обработана последовательность между элементами `'012345678'[-5] = '4'` и `'012345678'[8] = '7'`. Соответственно, в срез попадут только выделенные символы `'0123`**`4567`**`8'` с шагом 2. В результате чего получим строку `'46'` (см.рис.2).

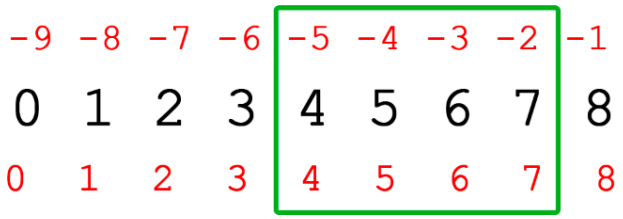


Рисунок 2. Срез при указании индексов разными способами индексирования

Кортежи

Самый простой тип последовательностей. С ним буквально нельзя делать ничего, кроме общих операций из таблицы выше.

Кортеж – последовательность объектов, которая определена единожды. То есть мы не можем переопределять уже добавленный в кортеж объект.

При сложении двух кортежей получается новый объект, содержащий элементы двух кортежей в добавленном порядке.

Операция += для кортежа работает аналогично записи `tup = tup + add_tup`, так как функция расширения объекта не применима к объектам неизменяемого типа данных. Другими словами создается НОВЫЙ объект, ссылкой на который будет переопределенная переменная. Аналогично работает оператор *=.

Кортежи очень полезны, когда нам нужно обработать массив данных при этом не изменив его.

Списки

Списки – наиболее часто используемый тип для хранения последовательностей. Он весьма удобен, так как позволяет сохранить как отдельные значения, так и сложные объекты в качестве своих элементов.

Обычно списки используют для хранения однотипных данных. Такой подход позволяет писать быстрые и удобные алгоритмы для обработки хранимых значений.

Поддерживаемые операторы

Операторы изменения объекта `lst += lst2` и `lst *= N` расширяют объект при этом не меняя ссылку на него.

Операторы проверки вхождения элемента в последовательность `val in lst` и `val not in lst` определяют входит или не входит элемент со значением *val* в список *lst.*

Методы работы со списками

`s.`**`append`**`(value)/s.`**`extend`**`(values)`

Методы расширения списка.

- *append* добавляет ОДИН элемент *value* в конец списка s.
- *extend* присоединяет список *values* в конец списка *s*.

Для добавления одного элемента рекомендуется использовать *append*, для добавления нескольких – *extend*.

Примеры.

```
s = [1, 2, 3]
s.append(5)        # [1, 2, 3, 5]
s.extend([2,4,6]) # [1, 2, 3, 5, 2, 4, 6]
```

`s.`**`remove`**`(value)`

Удаляет первое вхождение значения *value* из списка.

Если в списке нет искомого значения, будет возвращена ошибка «*ValueError: list.remove(x): x not in list*» или
«Ошибка значения: *list.remove(x)*: *x* отсутствует в списке».

Примеры.

```
s = [1, 2, 3]
s.remove(2)  # [1, 3]
s.remove(10) # ValueError
```

`s.copy()`

Метод возвращающий копию объекта.

Пример.

```
s = [1, 2, 6, 10]
sc = s.copy() # [1, 2, 6, 10]
```

ВАЖНО: возвращается неглубокая копия, то есть копируются ссылки на объекты оригинального списка. Поэтому, если в качестве элемента оригинального списка было значение изменяемого типа, то вернется ссылка на этот же объект.

Пример.

```
# пример работы с изменением объекта
s = [1, [1, 2], 3, 4]
sc = s.copy()
sc[1] += [2, 4] # меняем sc
print(s) # [1, [1, 2, 2, 4], 3, 4] - s меняется тоже
```

Для создания полного дубликата (глубокое копирование) можно воспользоваться методом *deepcopy()* из библиотеки *copy*.

`s.sort(reverse=True/False)`

Метод, сортирующий элементы в списке s. Также может производить сортировку в обратном порядке (при указании аргумента *reverse=True*).
С методами настраиваемых сортировок мы разберемся в одной из следующих глав.

ВАЖНО: изменяет список *s* в отличии от функции *sorted* для последовательностей.

Примеры.

```
s = [1, 3, 2, 10, 6, 2]
s.sort()                # s = [1, 2, 2, 3, 6, 10]
s.sort(reverse=True)    # s = [10, 6, 3, 2, 2, 1]
```

`s.reverse()`

«Переворот» списка, расположение элементов в обратном порядке. Результат аналогичен взятию среза `s[::-1]` за тем исключением, что при использовании `s.reverse()` объект останется прежним.

Пример.

```
s = [1, 5, 2, 8]
s.reverse()      # [8, 2, 5, 1]
```

`s.`**`insert`**`(index, value)`

Вставка значения *value* на позицию *index*. Все элементы, которые имели позицию бóльшую или равную значению *index*, сдвигаются на 1 вправо (индекс увеличивается на 1).

ВАЖНО: если указать значение *index*, выходящее за допустимый диапазон индексов для изменяемого списка, то в случае значения *index*, меньше допустимой левой границы, элемент будет добавлен в начало списка, в случае значения *index* больше длины последовательности – в конец.

Примеры.

```
s = [1, 2, 3]
s.insert(1, 10)     # [1, 10, 2, 3]
s.insert(0, 11)     # [11, 1, 10, 2, 3]
s.insert(-100, 8)   # [8, 11, 1, 10, 2, 3]
s.insert(20, 22)    # [8, 11, 1, 10, 2, 3, 22]
```

`s.`**`pop`**`(index)`

Удаляет элемент с индексом *index* и возвращает его в качестве результата. В случае указания недопустимого индекса возвращается ошибка «*IndexError: pop index out of range*» или «Ошибка индекса: указанный индекс удаляемого элемента выходит за допустимый диапазон».

Если не указывать значение *index*, будет удален последний элемент списка.

Примеры.

```
s = [1, 2, 3, 4, 5]
x = s.pop(3)          # s = [1, 2, 3, 5], x = 4
s.pop(100)            # IndexError

s = [1, 2, 3, 4, 5]
y = s.pop()           # s = [1, 2, 3, 4], y = 5
```

Строки

Необходимо понимать, что строковый тип данных неизменяемый. Поэтому всегда, когда мы получаем строку отличную от обрабатываемой, создается новый объект.

Также нужно понимать, что отдельный символ также является строкой.

```
>>>s[i] == s[i:i+1]
True
```

Поддерживаемые операторы

Операторы изменения объекта *s += sub* и *s *= N* работают аналогично записям *s = s + sub* и *s = s * N* соответственно, создавая новый объект и связывая старое имя переменной с созданным объектом.

Операторы проверки вхождения элемента в последовательность *sub in s* и *sub not in s* определяют является/не является ли строка *sub* подстрокой (частью) *s*.

Методы работы со строками

Строковый тип данных поддерживает достаточно большое количество методов. В контексте данного учебника мы не будем рассматривать их все, при желании с полной документацией можно ознакомиться в источнике [10]. Все указанные методы НЕ ИЗМЕНЯЮТ объект строки для которого вызваны.

`s.`**`find`**`(sub[, start[, end]])`
`s.`**`index`**`(sub[, start[, end]])`

Возвращает число – индекс первого вхождения слева подстроки *sub*, начиная с индекса *start* и заканчивая индексом *end*. Аргументы *start* и *end* – необязательные.
Для поиска справа используются аналоги – *rindex, rfind*.
 	
Разница *index* и *find*:
• если *find* не находит совпадение, результат равен -1,
• если *index* не находит совпадение – интерпретатор возвращает ошибку.
 	
Примеры.
`'adbefb'.find('bef')  # 2`
`'adbefb'.index('bef') # 2`
`'adbefb'.index('cef') # Error`
`'adbefb'.find('aaa')  # -1`

`s.`**`isdecimal`**`()`

Возвращает логическое значение – *True*, если все символы являются десятичными цифрами, *False* в обратном случае. Стоит заметить, что это символы, которые входят в категорию «*Number, Decimal Digits, Nd*» таблицы *Unicode* [11].

Примеры.

```
'12345'.isdecimal()      # True
'123asd321'.isdecimal() # False
```

`s.`**`isnumeric`**`()`

Возвращает логическое значение – *True*, если все символы являются числовыми символами, *False* в обратном случае. Дополнительно к символам из категории «*Number, Decimal Digits*» принимает символы из категории «*Other Number, No*» [12]

Примеры.

```
'12345'.isnumeric()      # True
'123asd321'.isnumeric() # False
```

`s.`**`isdigit`**`()`

Возвращает логическое значение – *True*, если все символы являются цифирными представлениями, *False* в обратном случае. Работает с подмножеством множества символов для *s.isnumeric*(), которые соответствуют цифрам от 0 до 9.

Примеры.

```
'12345'.isdigit()      # True
'123asd321'.isdigit() # False
```

Для обработки символьных последовательностей, где не может быть десятичных цифр, кроме '0123456789', подойдет любой из методов. Однако, стоит понимать, что данные методы обрабатывают разные наборы символов.

`s.`**`join`**`(strings)`

Возвращает строку – объединенные строки из последовательности *strings* через разделитель *s*.

ВАЖНО: в качестве аргумента метода могут быть только последовательности из строк, либо одна строка (которая будет интерпретирована как последовательность строк единичной длины).

Примеры.

```
'0'.join(('1', '223', 'abc')) # '102230abc'
'_'.join('hello')             # 'h_e_l_l_o'
```

`s.`**`replace`**`(old, new[, count])`

Возвращает строку – ***копию строки*** *s*, в которой заменяет подстроки *old* на строки *new count* раз. Исходная строка при этом не меняется. Замены происходят слева направо.
Если *count* не указан, заменяются все вхождения *old* на *new*.

Метод *replace* работает с непересекающимися подстроками, выбирая первое левое вхождение *new* в *old* перед заменой. Также можно сказать, что метод работает не рекурсивно.

Примеры:

```
'pppp'.replace('pp', 'p p')      # 'p pp p'
'123321'.replace('1', '202', 1)  # '20223321'
'Hello'.replace('sh', 'pr')      # 'Hello'
```

`s.`**`split`**`(sep=None, maxsplit=-1)`

Возвращает список из подстрок, полученный путем разделения строки *s* по разделителю *sep*. По умолчанию делит строку по пробельным символам.
Если *split* встречает два разделителя *sep* подряд, то возвращает пустое слово, которое соответствует положению этих разделителей.
При указании параметра *maxsplit* разделяет по первым *maxsplit* разделителям.

После разделения строки по указанному разделителю *sep* можно восстановить строку с помощью записи *sep.join(s.split(sep))*. Данный метод восстановления строки не работает при работе с неуказанным разделителем, то есть запись *sep.join(s.split())* не восстановит начальное значение строки *s*.

Для разделения строки справа, используется аналог *rsplit*.

ВАЖНО: метод *s.split()* без указания разделителя делит строку по пробельным группам, то есть в качестве разделителя используется группа пробельных символов (например, группа пробелов или переносы строк). В таком случае не получаются пустые строки, как, например, в случае с указанием в качестве разделителя одиночного пробела.

Примеры.

```
'123123'.split('2', maxsplit=1) # ['1', '3123']
'abcdabcd'.split('dab')         # ['abc', 'cd']
'два  пробела'.split()    # ['два', 'пробела']
'два  пробела'.split(' ') # ['два', '', 'пробела']
'пр дл rs'.rsplit(' ', maxsplit=1) # ['пр дл', 'rs']
```

`s.strip([chars])`

Возвращает строку – ***копию строки*** *s*, в которой удаляет все символы *chars* с концов строки *s* до первого несовпадения с последовательностью *chars*. Для удаления ненужных символов справа или слева используются аналоги – *rstrip* и *lstrip* соответственно.

Примеры.

```
'123abcdef46'.strip('1234567890')  # 'abcdef'
'13323asv123'.rstrip('1234567890') # '13323asv'
'13323asv123'.lstrip('1234567890') # 'asv123'
```

`s.lower()/s.upper()`

Возвращает строку – ***копию строки*** *s*, в которой все символы преобразованы в нижний/верхний регистр.

Примеры:

```
'fvbd123sDf'.lower()  # 'fvbd123sdf'
'fvbd123sDf'.upper()  # 'FVBD123SDF'
```

`s.count(sub)`

Возвращает число – количество непересекающихся подстрок *sub* в строке *s*.

Примеры:

```
'222mm2222'.count('22')    # 3  '222mm2222'
'123a12nb12'.count('12')   # 3
'helloworld'.count('bye')  # 0
```

Так же существует пара функций, которая позволяет работать с символами и кодировочной таблицей *Unicode* [13].

`chr(num)`

Возвращает строку – символ по его номеру в кодировочной таблице *Unicode*.

Примеры.

```
chr(123) # '{'
chr(32)  # ' ' – пробел
```

`ord(symbol)`

Возвращает число – номер символа в кодировочной таблице *Unicode*.

Примеры.

```
ord('a') # 97
ord(' ') # 32
```

ВАЖНО: сравнение символов происходит исходя из их позиции в таблице *Unicode*. Больше номер – больше символ.

ВАЖНО!

При написании алгоритмов преобразующих строки необходимо помнить, что результаты работы методов нужно сохранять в качестве нового объекта. Например, при переносе псевдокода из следующего задания на язык программирования *Python*, необходимо сохранять результат преобразования.

Также существует отдельный класс символов – графемы. Это специальные знаки, которые размещаются поверх других символов. Получаемые таким образом символы записываются как несколько символов.

Например, ḍ – это совмещенные символ d и графема ̣. Для обработки таких символов функции *ord* и *chr* не применимы.

Пример.

Исполнитель Редактор получает на вход строку цифр и преобразует её.

Редактор может выполнять две команды, в обеих командах v и w обозначают цепочки символов.

1. заменить (v, w)
2. нашлось (v)

Первая команда заменяет в строке первое слева вхождение цепочки v на цепочку w. Если цепочки v в строке нет, эта команда не изменяет строку. Вторая команда проверяет, встречается ли цепочка v в строке исполнителя Редактор. На вход приведённой ниже программе поступает строка, состоящая из 107 букв X. Какая строка получится после выполнения программы?

```
НАЧАЛО
ПОКА нашлось(XXX) или нашлось(ZYX) или нашлось(ZXX)
  заменить(XXX, ZZ)
  заменить(ZYX, X)
  заменить(ZXX, Y)
КОНЕЦ ПОКА
КОНЕЦ
```

Код на языке Python

```
s  = 'X'*107
while 'XXX' in s or 'ZYX' in s or 'ZXX' in s:
    s = s.replace('XXX', 'ZZ', 1)
    s = s.replace('ZYX', 'X', 1)
    s = s.replace('ZXX', 'Y', 1)
print(s)
```

Экранирование

Нередко встречается необходимость ввести в строке спецсимвол, начиная от кавычки, которая используется для ограничения начала и конца строки, заканчивая символами которых нет на клавиатуре (перенос строки). Для таких случаев используется экранирование или *escape*-последовательности [14].

В качестве экранирующего символа используется обратный слеш «\».

Некоторые наиболее часто употребляемые escape-последовательности.

***Escape*-последовательность**	**Значение**
`\newline`	Обратный слеш игнорирует перенос строки
`\\`	Обратный слеш
`\'`	Одинарная кавычка
`\"`	Двойная кавычка
`\n`	Перевод на строку (*LF*)
`\r`	Возвращение каретки (*CR*)
`\t`	Горизонтальная табуляция (*TAB*)
`\N{name}`	Символ с именем name в *Unicode*
`\uxxxx`	Символ с 16-битным значением xxxx в *Unicode*
`\Uxxxxxxxx`	Символ с 32-битным значением xxxxxxxx в *Unicode*

Интересный факт.

Привычный нам разрыв строки на самом деле является комбинацией двух символов – «Возвращение каретки» и «Перевод на строку». При использовании *escape*-последовательности «\n» *Python* делает сразу два действия – переносит курсор на новую строку и передвигает каретку в начало строки. Исторически же это последовательность *\r\n* и пошла она с использования печатных машинок, при работе на которых для начала печати с новой строки передвигали печатающий вал, который при передвижении до левого края прокручивался и устанавливал бумагу в положение, соответствующее новой строке.

Символ *\r* можно использовать, чтобы печатать поверх выведенного текста, заменяя уже выведенные символы на новые.

Пример (*time.sleep(n)* – функция, осуществляющая задержку *n* секунд).

```
from time import sleep
print("Loading 0%...", end='')
sleep(1)
print("\rLoading 1%...", end='')
```

Одной из частых проблем при работе с экранированными символами является открытие файла по абсолютному адресу, где адрес представлен, как строка *'имя_диска:\путь\до\файла.txt'*. Дело в том, что такая строка будет считать, что экранированы символы «*п*», «*д*» и «*ф*».

Чтобы этого избежать, необходимо либо продублировать символ обратного слеша *'имя_диска:\\путь\\до\\файла.txt'*, либо дописать префикс *r (*от англ. *raw* – необработанный) перед строкой ***r****'имя_диска:\путь\до\файла.txt'*.

При использовании префикса *r* стоит помнить, что строка не может оканчиваться нечетным количеством обратных слешей, так как в таком случае последний обратный слеш будет экранировать кавычку, закрывающую строку.

Либо можно заменить все обратные слеши на прямые, например, *'имя_диска:/путь/до/файла.txt'*

Преобразование строки в список/кортеж

Также строку можно преобразовать в список или кортеж отдельных символов.

```
>>>s = '123455'
>>>list(s)
['1', '2', '3', '4', '5', '5']


>>>s = '112345'
>>>tuple(s)
('1', '1', '2', '3', '4', '5')
```

Форматирование чисел при выводе

f-строки – очень удобный способ форматирования строк, который позволяет удобно добавлять в строку значения переменных и форматировать их. Поддержка *f*-строк начинается с версии *Python* 3.6, выпущенной в 2015 году, поэтому шанс встретить версию интерпретатора без поддержки *f*-строк крайне невелик.

Рассмотрим лишь некоторые особенности использования *f*-строк.

Количество знаков до и после запятой

```
f'{x[:[0][count].[precision]}'
```

На 0 указывается, если необходимо дополнить выводимое число нулями до количества *count*. Значение *precision* отвечает за количество знаков после запятой.

Примеры.

```
>>>x = 3/7
>>>f'{x:.5}'
'0.42857'
>>>f'{x:10.4}'
'    0.4286'
>>>f'{x:08.3}'
'0000.429'
```

Перевод целого числа в 2, 8 и 16 системы счисления.

```
f'{x[:[[0]count]type]}'
```

На месте *count* указывается необходимое количество символов, которое должно занимать выводимое значение. 0 указывается в случае, если нужно вывести незначащие нули до длины *count*. В качестве *type* может стоять одно из следующих значений:

Тип	**Значение**
b	Двоичный формат
d	Десятичный формат
o	Восьмеричный формат
x	Шестнадцатеричный формат (нижний регистр)
X	Шестнадцатеричный формат (верхний регистр)

Говоря иначе доступны следующие форматы:

`f'{x}'`

Выводит десятичное число без форматирования.

```
>>> f'{123}'
'123’
```

`f'{x:<type>}'`

Выводит число в указанном формате.

```
>>> f'{123:b}'
'1111011'
>>> f'{123:X}'
'7B'
```

`f'{x:<count><type>}'`

Выводит число в формате *type* длиной *count* символов.

```
>>> f'{12:6b}'
'  1100'
>>> f'{123:5X}'
'   7B'
```

`f'{x:0<count><type>}'`

Выводит число в формате *type* из *count* символов, в котором при необходимости дописаны ведущие нули.

```
>>> f'{12:60b}'
'001100'
>>> f'{123:50X}'
'0007B'
```

Если не указывается формат вывода *type*, то по умолчанию принимается десятичный.

Если *count* или *type* являются вычислимыми значениями или значениями переменных, то переменная или выражение берутся в фигурные скобки.

```
>>> x = 15
>>> count = 20
>>> f'{x:0{count}}'
'00000000000000000015'
```

Операторы переопределения

Так как последовательности поддерживают операторы + и *, они также поддерживают и операции += и *=.

При работе со списками (и другими объектами изменяемого типа) важно понимать, что хоть ссылка на объект и остается прежней, эти операторы являются операторами присваивания.

Важность этого поведения мы можем увидеть при работе с кортежем изменяемых объектов.

Пример.

```
a = ([1, 2], [2, 3])
```

В этой строке мы создали кортеж, который хранит две ссылки на объекты списков. Данные ссылки нельзя менять, то есть пара объектов должна всегда быть одними и теми же объектами. Попробуем дополнить любой из объектов.

```
a[0] += [4, 5]
```

Интерпретатор вернет ошибку

«*TypeError: 'tuple' object does not support item assignment*»

«*Ошибка типа: объект ‘кортеж’ не поддерживает операцию присваивания элементов*».

Однако, если мы обратимся к методам списка, которые его расширяют:

```
a[0].append(3)
a[0].extend([4, 5])
```

То такой ошибки не будет, потому что нет ошибки присваивания и мы обращаемся к методу объекта, что не запрещено для объектов внутри кортежа.

Для иллюстрации оператор += можно заменить на эквивалентную запись с использование функции *iadd* модуля *operator*.

```
import operator
x = [1, 2]
# Эквивалентная запись для x += [4, 5]
x = operator.iadd(a, [4, 5])
```

Объект *x* будет иметь прежний *id*. Однако, исходя из того, как работает оператор присваивания, мы понимаем, что *operator.iadd* возвращает ссылку на преобразованный объект *x*, что считывается интерпретатором как переприсваивание или переопределение объекта, на который ссылается переменная.

Итераторы

При работе с итераторами можно выделить две базовые сущности – итерируемый объект (*iterable object*) и собственно итератор (*iterator*).

Если коротко и несколько упрощенно, то отличия следующие. Итерируемый объект – это такой объект, элементы которого можно перебрать. Например, итерируемым объектом может быть любой объект-последовательность.

Итератор – это специальный объект, который перебирает элементы итерируемого объекта.

Чтобы из итерируемого объекта создать итератор, необходимо передать его в функцию *iter*(). После чего можно получать значения из итерируемого объекта с помощью функции *next*() подавая ей на вход итератор с предыдущего шага.

Пример.

```
>>> it = iter('abc')
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    next(it)
StopIteration
```

В рассмотренном примере можно заметить, что функция *next*() последовательно вывела все элементы строки *'abc'* и при следующем обращении вернула исключение *StopIteration*.

И в этом состоит одно очень важное отличие итераторов от итерируемых объектов. Итератор может обратиться к каждому элементу только один раз в порядке слева направо. При работе же с итерируемым объектом мы всегда можем обратиться к элементу или проверить его наличие.

Рассмотрим данное утверждение на примере работы оператора *in*.

Оператор *in* для последовательностей (итерируемых объектов) и итераторов работает схожим образом – проверяет последовательно элементы слева направо и, если находит элемент с искомым значением, возвращает *True*, иначе *False*.

Пример.

```
>>> obj = [1, 2, 3, 4, 5]
>>> it = iter(obj)
>>> 3 in obj
True
>>> 3 in it
True
```

Видим идентичные результаты. Значит ли это, что итератор и итерируемый объект равнозначны? Хочется ответить, что да. И в этом заключается очень частая ошибка тех, кто забывает результат в виде итератора преобразовать в последовательность.

При повторном таком обращении мы увидим отличные поведения для итератора и списка.

```
>>> 3 in obj
True
>>> 3 in it
False
```

В чем отличие первого вызова от второго? Как было сказано ранее, оператор *in* обрабатывает элементы слева направо. Условно работу можно рассмотреть как пошаговую следующим образом.

№ шага	**Список**	**Итератор**	**Результат сравнения**
1	`obj[0] == 3 # 1`	`next(it) == 3 # 1`	False
2	`obj[1] == 3 # 2`	`next(it) == 3 # 2`	False
3	`obj[2] == 3 # 3`	`next(it) == 3 # 3`	True !!!

То есть после первого срабатывания оператора *in* список в *obj* не изменился, однако теперь *next*(*it*) вернет нам 4. Поэтому при следующем вызове *next*(*it*) итератор продолжит с места, где закончил в прошлый раз – на значении 4.

Тут же важно отметить, что итератор не хранит все значения, которые возвращаются путём вызова функции *next*(), он получает их по мере обращения. То есть запоминает место, где закончил вывод и при следующем обращении к *next*() возвращает следующий элемент.

При работе на *python* мы будем часто обращаться к функциям и методам, которые возвращают именно итератор. Поэтому это отличие для нас очень значимое.

Функция map

Функция, которую используют все, но не все задумываются, как она работает.

```
map(function, iterableObject[,io2[,io3[, ...])
```

Функция *map* возвращает итератор, каждый элемент которого является результатом применения функции *function* к элементу итерируемого объекта *iterableObject* или к параллельным наборам перечисленных через запятую итерируемых объектов. Во втором случае перебираемые группы значений передаются как отдельные значения (подробнее в одной из следующих глав).

Также отдельно следует упомянуть важную особенность работы с результатами работы функции *map*.

Если мы присвоим переменной результат *map*, то переменная будет ссылаться на итератор! Поэтому для возвращения всех обработанных функцией *map* элементов необходимо преобразовать результат в одну из допустимых последовательностей, например, кортеж.

Пример.

```
>>> map(str, [123,321,1111,101])
<map object at address>
>>> tuple(map(str, [123,321,1111,101]))
('123', '321', '1111', '101')
```

Если же мы присваиваем нескольким переменным результат работы *map*, то количество определяемых значений переменных должно соответствовать количеству значений, возвращаемых итератором.

Так, если функция *map* обрабатывает X значений, а слева от знака присваивания будет Y ($Y \neq X$) переменных, то *Python* вернет одну из двух ошибок распаковки:

- «*ValueError: not enough values to unpack (expected X, got Y)*» или «Ошибка значения: недостаточно значений для распаковки (ожидалось X, вернулось Y)»,
- «*ValueError: too many values to unpack (expected X)*» или «Ошибка значения: слишком много значений при распаковке (ожидалось X)»

Пример распаковки.

Необходимо ввести три целочисленных значения, заданных в одной строке через пробел.

Вариант 1:

1. Мы уже знаем, что для разбивки строки по пробелу есть функция *s.split()*, которая по умолчанию разбивает строку по пробельным символам.

```
str_values = input().split()
```

2. Теперь переменным x, y, z необходимо присвоить числовые значения, соответствующие введенным строковым значениям.

   Способ 1.

```
x = int(str_values[0])
y = int(str_values[1])
z = int(str_values[2])
```

   Способ 2.

```
int_iter = map(int, str_values)
x, y, z = next(int_iter), next(int_iter), next(int_iter)
```

Вариант 2:

```
x, y, z = map(int, input().split())
```

Вариант №2 является сокращенной записью алгоритма из варианта №1. Однако если у нас будет введено больше значений через пробел, то вариант №2 вернет ошибку. Если меньше, то не сработает и первый.

Поэтому вариант, который работает правильно, будет такой.

```
str_values = input().split()
if len(str_values) == 3:
    x, y, z = map(int, str_values)
```

На ЕГЭ таких проверок делать не надо, однако при написании более сложных алгоритмов необходимо учитывать возможность некорректного ввода.

Чтение из файла

open(filename, mode)

Создание объекта для чтения текстового файла с путём *filename*. *mode* выставляется в зависимости от режима работы с файлом. По умолчанию файл открывается для чтения.

```
file_obj = open(filename)
```

Возвращаемый в *file_obj* объект так же является итератором, то есть ссылается на место в файле, на котором была завершена последняя операция считывания данных. При последовательном обращении к итератору через *next()* возвращаются строки файла.

next(file_obj) эквивалентно вызову *file_obj.readline()* (см.далее).

Пример – открытие файла для чтения.

```
file = open('27-A.txt')
```

f.**read**(count)

Считывание следующих *count* символов в текстовом файле.
По умолчанию считывается весь файл.
Пример – файл *test.txt* содержит строку `'abcdefg'`.

```
file = open('test.txt')
s = file.read(3) # 'abc'
s = file.read(2) # 'de'
s = file.read(5) # 'f'
s = file.read(5) # ''
```

f.**readline**()

Чтение следующей несчитанной строки до переноса строки.
Если вызвать метод после считывания всего файла, будет возвращена пустая строка.

ВАЖНО: метод *readline*() возвращает строку с последним символом \n, кроме случая, когда это последняя строка.

f.**readlines**()

Чтение всех несчитанных строк, разделенных переносом строки.

ВАЖНО: каждая строка в полученном списке также, как в случае с методом *readline*(), оканчивается сносом строки.

`f.close()`
Удаление объекта для работы с файлом из памяти. Данное действие закрывает файл. Тем самым позволяет открыть этот файл другим потоком. Особенно актуально закрывать файл, открытый для дозаписи.

`with open(filename, flag) as f:`
`  # обработка файла`
`# файл закрыт без f.close()`

«Безопасный» с точки зрения работы с памятью метод чтения информации из файла. После выхода из блока команд для *with* объекта для работы с файлом не будет. И файл закроется автоматически.

При написании простых программ, особенно в рамках ЕГЭ, нет значительной разницы, каким образом вы работаете с файлом – удаляете ли объект после обработки всего файла или нет. После завершения работы скрипта все объекты в памяти все равно перестанут существовать.

Проблемы незакрытых файлов:

- утечка памяти для хранения объектов,
- блокировка текущей программой файла, например, если он открыт для записи,
- потеря данных в случае, если программа аварийно завершилась.

Поэтому очень важно всегда контролировать закрыт ли обработанный файл.

Также лучше использовать функции для последовательного считывания данных. Тогда при обработке файлов большого размера не будет возникать ошибок, связанных с переполнением памяти.

Так, следующие два примера хоть и делают одно и тоже, однако вариант, где мы используем метод *readlines* требует больше памяти и при работе с большими файлами потребует эту самую память на первоначальное считывание всех строк файла.

Не нужно выделять память на все строки из файла	Нужно выделять память на все строки из файла
`with open('test.txt') as f:` `  data = list(map(int, f))`	`with open('test.txt') as f:` `  data=list(map(int, f.readlines()))`

Функции

Что такое функции

При создании сложных алгоритмов зачастую приходится иметь дело с одинаковыми участками кода, которые реализуют один и тот же служебный алгоритм. Такие участки кода принято оформлять, как функции.

Функцию можно рассматривать, как именованный участок кода, который вызывается для исполнения закодированного в нем алгоритма. Также существуют анонимные (неименованные) функции, о них поговорим ниже.

Например, если в ходе нашего алгоритма необходимо при выводе в нескольких местах форматировать строку в формате *"HH:MM:SS"*, то мы можем описать функцию *format_time_to_str(h, m, s)* и вместо описания условий форматирования каждый раз вызывать описанную функцию, которая вернет нужную строку.

Такой подход локализовать ошибки, ведь при описании функции мы описываем алгоритм один раз и, в случае обнаружения ошибок его работы, исправляем ошибки в одном месте нашей программы, а не везде, где этот алгоритм был необходим.

Функции в *Python* можно условно разделить на те, которые возвращают значение, и те, которые его не возвращают. Первые применяются для получения результатов обработки значение, вторые для выполнения отдельных действий.

Примеры функций, которые не возвращают значение.

- Записать в файл журнала сообщение,
- Отправить сообщение в чате,
- Показать справку по команде.

Примеры функций, которые возвращают значение.

- Найти корни квадратного уравнения,
- Найти количество путей на графе по матрице смежности,
- Найти произведение матриц.

Функция, как и что угодно в *Python*, является объектом – *PyFunctionObject*. Условно описание функции можно разделить на следующие составляющие:

- имя функции,
- параметры, принимаемые на вход,
- исполняемый код,
- аннотация обрабатываемых и возвращаемых значений,
- документация функции.

Правила именования функций – неразрывная строка из латинских букв в верхнем и нижнем регистрах и символов нижнего подчеркивания. Традиционно имена функций, как и имена переменных, записываются в «змеином» стиле – строка в нижнем регистре, слова в которой разделены знаком подчеркивания.

Пример описания функции:

```
def func_name(x, y, z):
  # блок команд
```

У такой функции имя *func_name*, и она принимает на вход 3 значения, для которых заданы имена *x, y, z*.

Возвращение значений

Для возвращения значения, являющимся результатом работы функции, необходимо использовать оператор *return*.

```
def simple_function():
  return 2 + 2
```

ВАЖНО: после выполнения оператор *return* возвращает управление тому блоку команд, в котором была вызвана функция. То есть код, записанный после оператора *return* выполняться не будет.

```
def simple_function():
  return 2 + 2
  # этот код выполняться не будет
  x = 5*10
```

Функция может не возвращать значение. Тогда результатом выполнения функции будет объект *None*.

```
def print_10():
  print(10)
x = print_10() # None
```

Параметры

Так как выполнение одного и того же алгоритма, который обрабатывает одинаковые значения, редко бывает полезным. Обычно гораздо проще найти значение один раз и использовать вычисленное значение далее.

Например, нет смысла постоянно находить результат вычисления значения выражения $3x^2+10x-20$ для $x = 1$. Гораздо полезнее, при частом обращении к данному выражению, описать функцию *f(x)*, которая будет вычислять значение выражения для любого числового значения *x*.

Пример объявления функции для вычисления выражения $3x^2+10x-20$.

```
def calc_expression(x):
  return 2*x**2 + 10*x - 20
print(calc_expression(5)) # 80
```

Стоит отметить, что есть еще один термин – аргументы. Аргументы – это значения, передаваемые на вход функции. Так, в предыдущем примере, аргументом является значение 5.

Также аргументы иногда называют фактическими параметрами.

Значения параметров по умолчанию

Также мы можем определить значения, которые будут приняты функцией, если мы не зададим их значение.

Например, мы можем определить функцию для нахождения суммы цифр в десятичной записи числа, представленного в виде строки *s* в системе счисления с основанием *base*. По умолчанию будем считать, что основание системы счисления равно 10.

```
def sum_digit(s, base=10):
  str_base10 = s
  if base != 10:
    x = int(s, base)
    str_base10 = str(x)
  return sum(map(int,str_base10))
```

Все параметры, значения которых задаются по умолчанию, описываются последними. При этом, если количество передаваемых функции значений меньше определенного для нее количества параметров, то по умолчанию берутся параметры, начиная от последнего описанного.

Например, если имеем такое объявление функции

```
def test(a, b, c=10, d=15):
  pass # оператор пустого блока команд
```

В случае передачи трех значений в качестве аргументов функции значение *d* будет принято по умолчанию. Если же мы передадим такой функции меньше двух значений, то интерпретатор вернет ошибку, так как все параметры должны быть определены.

При вызове функции можно задавать значение параметров по имени.

Например, для функции *test()* можно сделать так

```
test(b = 10, a = 5, d = 11)
```

Однако обычно значения параметров без значений по умолчанию указывают в том порядке, в котором они заданы при объявлении функции. В том числе поэтому их называют позиционными.

Запрещается описывать параметры со значением по умолчанию перед описанием параметров без таковых.

Пример, как нельзя описывать параметры функции.

```
def bad_example(a, b = 10, c):
  pass
```

ВАЖНО: объекты, на которые будут ссылаться имена аргументов, создаются ОДИН РАЗ при инициализации функции. Это очень важно учитывать, если в качестве значения по умолчанию мы хотим определить значение изменяемого типа (например, список).

Пример.

```
def append_to_list(x, ext_list = []):
  ext_list.append(x)
  return ext_list
```

Казалось бы, при передаче единственного аргумента мы будем получать список, содержащий переданное значение. Однако при вызове функции с одним значением изменяется объект, на который ссылается параметр *ext_list*. Убедиться в изменении значений по умолчанию можно вызвав служебный метод *__defaults__* объекта функции или проанализировав возвращаемое функцией значение.

```
print(append_to_list.__defaults__) # ([], )
append_to_list(4)
append_to_list(6)
print(append_to_list.__defaults__) # ([4, 6, 8],)
```

Аннотация

Аннотация типов данных в *Python* [15] является больше средством для удобства разработки, чем указанием интерпретатору, какие данные будут сохранены в переменных или переданы в качестве аргументов при вызове функции. На сегодня существуют анализаторы аннотированного кода, например [16], которые указывают на несоответствие присваиваемого значения переменной типу, указанному в аннотации. Однако при запуске программы *Python* не проверит данные нюансы и вернет ошибку только на этапе применения не существующих операций для типа данных.

Значения переменных и параметров аннотируются следующим образом:

```
variable_or_parameter: expression
```

Строго говоря, можно указывать не только тип, но и описание переменной. Однако в основном аннотацию используют именно для указание ожидаемого типа данных для значения переменной или параметра.

Чтобы описать возвращаемый функцией результат, используется запись:

```
def func_name() -> expression
```

Зачем же аннотировать ожидаемые значения, если *Python* все равно их проигнорирует при исполнении программы?

Такой подход повышает удобство разработки, так как большинство IDE поддерживают, например, вызов списка методов для переменных с определенным типом.

Рисунок 1. Пример подсказки для результата неаннотированной и анотированной функций

Также мы можем указать один из ожидаемых типов с помощью типа объединения *Union* из библиотеки *typing*.

```
def square(x: Union[int, float]) -> Union[int, float]:
    return x*x
```

С версии 3.10 объедининие типов может быть осуществленно через оператор |.

```
def square(x: int | float) -> int | float:
    return x*x
```

Для этого нет необходимости подключать отдельный модуль.

Документирование

Документирование функций, как и аннотация, позволяет быстрее понимать, как работать с функцией. Если аннотация, в основном, применима для описания типов обрабатываемых значений, то документирование – это описательная часть, которая раскрывает зачем нужна функция.

Документирование функций в *Python* происходит согласно *docstring*-соглашению [17]. Для этого используется формат описания в виде разметки *reStructuredText* [18], рекомендованный *PEP*-287 [19].

Описание работы функции записывается после её объявления в тройных кавычках.

```
def test():
    '''
    My first doc
    '''
    pass
```

Теперь *IDE*, обрабатывающий документированные функции, при наведении на функцию будут показывать написанный текст.

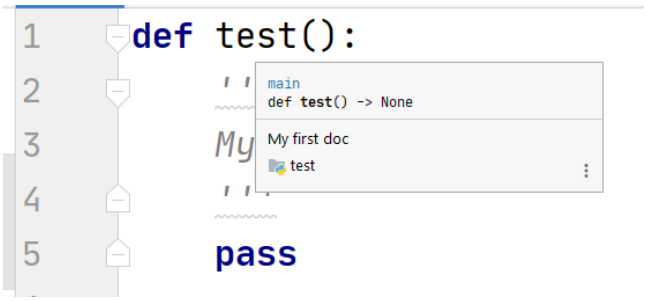


Рисунок 2. Пример всплывающего окна с текстом документации

Между абзацами в *docstring*-формате оставляется пустая строка. То есть между концом предыдущего абзаца и началом следующего ставить два переноса строки. Одиночный перенос строки используется для форматирования *docstring*.

Например, следующая документация к функции будет определена как двустрочная.

```
def test():
    '''
    My first very very
    very long string

    Second string
    '''
    pass
```

Во всплывающем окне такой текст будет выглядеть, как на рисунке ниже.

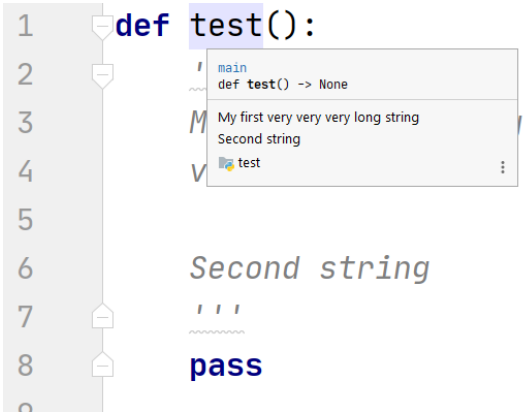


Рисунок 3. Пример вывода многострочной документации

Для описания параметров используется следующая разметка:

```
:param имя_параметра: описание
```

Описание возвращаемого результата с помощью `:return: описание`.

Пример. Функция для нахождения расстояния от точки до начала координат.

```
def distance(x:int | float,y: int | float) -> int | float:
    '''
    Calculate distance from origin
    to point with coordinate (x, y)

    :param x: Abscissa of point
    :param y: Ordinate of point
    :return: Distance origin to (x, y)
    '''
    return (x*x + y*y)**0.5
```

Теперь, при использовании этой функции, мы сможем увидеть подробное описание данной функции, что позволит нам не тратить время на изучение её кода (см.рис.4).

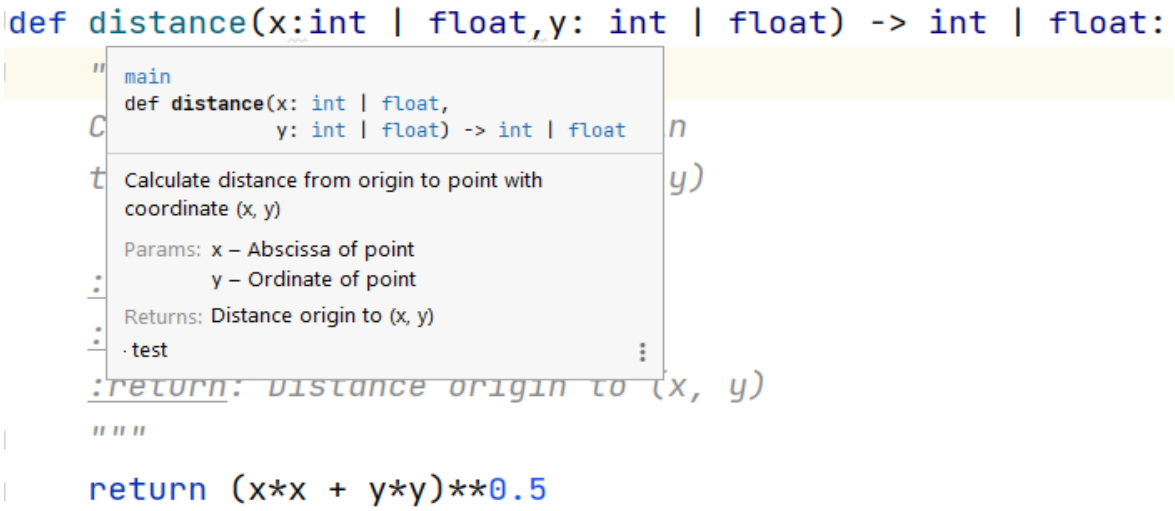


Рисунок 4. Пример документированной функции

Распаковка аргументов

Что же из себя представляют параметры и каким образом значения аргументов передаются в функцию.

Список и словарь аргументов

Как мы выяснили ранее, значения аргументов могут быть переданы как позиционно, так и по ключу. Поэтому любая функция принимает два объекта – список значений, переданных позиционно, и словарь значений, переданных по ключу.

Чтобы проиллюстрировать данное утверждение, можно провести распаковку переданных значений в передаваемые список и словарь. Подробнее про распаковку и словари поговорим в других главах учебника.

```
def test(*argv, **kwagrv):
  print(argv, kwargv)
```

В данной функции переменная *argv* будет ссылаться на список значений, переданные позиционно, *kwargv* будет хранить пары ключ-значение для всех значений, переданных по ключу. Например, при вызове следующей функции получим такие объекты.

```
test(4, '123z', [1,2], g=10, st='123asd')
# (4, '123z', [1, 2]) {'g': 10, 'st': '123asd'}
```

И такие две структуры всегда передаются при вызове функции. Далее проверяется, соответствуют ли переданные значения набору параметров. И, если соответствуют, то происходит соотнесение элементов переданных списка и словаря именам параметров.

Пример (корректная передача значений).

```
def test2(a,b,c,d):
  print(a+b+c+d)
test2(2, 6, c=5, d=15)
```

На вход функции *test2* подается два значения по позиции (2 и 6) и два значения по ключу (*c*=5 и *d*=10).

По позиции:
0. $a = 2$

1. $b = 6$

По ключу:
$c = 5$, $d = 10$
Передача значений прошла корректно, все обязательные параметры определены, нет противоречий по переданным значениям.

Пример (некорректная передача значений).

```
def test2(a,b,c,d):
  print(a+b+c+d)
test2(2, 6, 8, d=15, c=10)
```

По позиции:

0. $a = 2$
1. $b = 6$
2. $c = 8$

По ключу:

$c = 10$, $d = 15$

Видим противоречие, позиционно значение параметра *c* определенно, как 8, по ключу – как 10. Одна переменная не может ссылаться сразу на два значения, поэтому при таком вызове функции *test2* интерпретатор вернет ошибку «*got multiple values for argument 'c'*» или «получено несколько значений для аргумента *'c'*».

Также ошибка будет, если передается больше значений, чем описано в параметрах функции. Если же необходимо передать неограниченное количество значений аргументов, то можно определять оставшиеся позиционные параметры с помощью оператора *, параметры, передаваемые по ключу – с помощью оператора **.

Так следующая функция *other_pars* принимает на вход два позиционных аргумента *x, y*, один аргумент по ключу *kw* и все остальные параметры в список **other_pos*, и ***other_kw*.

```
def other_pars(x, y, *other_pos, kw, **other_kw):
  pass
```

Стоит отметить, что как только мы определили параметр через оператор *, все остальные параметры, описанные после него, могут быть переданы только по ключу.

Строго позиционные параметры и передача только по имени

Также, начиная с версии *Python 3.8*, мы можем определить, какие аргументы будут переданы только позиционно, какие только по ключу, а какие могут быть переданы любым способом. Данное поведение регламентируется соглашением *PEP-0570* [20].

```
def имя_функции(позиционные, /, любые, *, по ключу):
  pass
```

Обращение к функциям

Помимо обычного вызова функции в коде программы в месте, где требуется исполнить описанный в ней алгоритм. Существуют еще несколько распространенных методов.

Лямбда-выражения

Лямбда-выражение – это функция, которая объявлена в месте, где она должна быть вызвана. В языке *Python* написать лямбда-выражение весьма затруднительно из-за синтаксических особенностей его объявления. Однако, сказать, что лямбда-выражение не является функцией, нельзя, так как для него создается объект *PyFunctionObject*.

Примерно такое же суждение можно найти на страницах документации: «*Small anonymous functions can be created with the lambda keyword. Lambda functions can be used wherever function objects are required. They are syntactically restricted to a single expression*.»

«Небольшие безымянные функции могут быть созданы с помощью ключевого слова *lambda*. Лямбда-функции можно использовать везде, где требуются объекты-функции. Синтаксически они ограничены одним выражением.»

Синтаксис объявления лямбда-выражения.

```
lambda набор_параметров: выражение
```

Данное выражение возвращает объект, соответствующий функции с параметрами *par_list*, которая вычисляет значение выражения *expression*. Лямбда-выражения очень удобно использовать с функциями, обрабатывающими итераторы. Например, вместе с *map*.

Пример 1. Возвести все элементы списка в квадрат.

```
nums = [-10, 5, 11, -53]
sq = list(map(lambda x: x **2, nums))
# [100, 25, 121, 2809]
```

Помимо уже известных нам операций мы видим на месте первого аргумента функции *map* не известную нам функцию, а лямбда-выражение. То есть такой код преобразует все элементы списка *nums* по заданному выражению и сохранит результат в *sq*.

Работа с несколькими итераторами и map

Развернем информацию о том, как работает функция *map* с несколькими итерируемыми объектами.

Функция *map* параллельно достает по одному элемента из каждого итератора и подает полученные значения на вход функции, указанной в качестве первого аргумента.

Предположим, что у нас есть 3 итератора, которые возвращают в результате своей работы разное количество значений (см.рис.5).

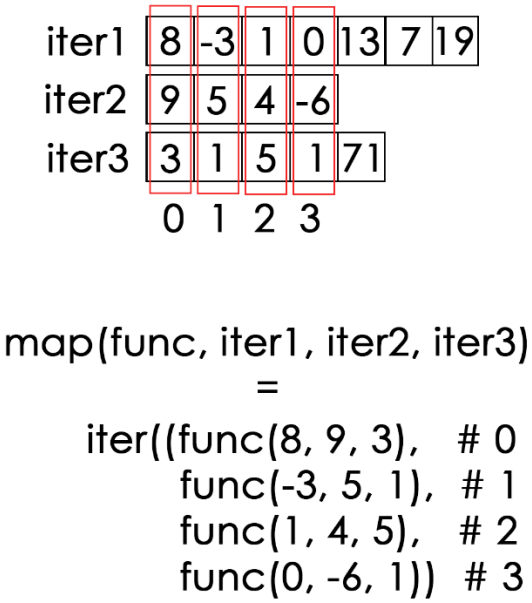


Рисунок 5. Пример работы функции map с несколькими итераторами

Пример. Найти сумму произведений пар смежных элементов списка.

```
list_ex = [2, 1, 5, 4, 8, 6]
```

Для списка *list_ex* нам необходимо вычислить сумму произведений

$$2 \cdot 1, 1 \cdot 5, 5 \cdot 4, 4 \cdot 8, 8 \cdot 6$$

На этом примере рассмотрим использование функции *map* для нескольких итерируемых объектов.

```
# находим все пары
pairs = map(lambda a, b: (a, b), list_ex, list_ex[1:])
# находим все произведения
prods = map(lambda p: p[0]*p[1], pairs)
# выводим максимум
print(max(prods))
```

Важно помнить, что на вход функции *func* передается количество аргументов, равное количеству переданных *map* итерируемых объектов.

Задача.

Строка состоит не более чем из 10^6 символов и содержит только заглавные буквы латинского алфавита E, G, K. Определите максимальное количество идущих подряд символов, среди которых сочетания символов KEGE повторяются не более двух раз [21].

Решение.

Для строки EGK**KEGE**GKGE**KEGE**HKEG**KEGE**KKK**KEGE**, например, мы можем взять подстроки EGK**KEGE**GKGE**KEGE**H**KEG** или **EGE**GKGE**KEGE**HKEG**KEGE**KKK**KEG.** Или другими словами три подстроки, разделенные подстроками *KEGE* и ограниченные символами *EGE* и *KEG*, если они находятся в середине строки.

Поэтому разобьем строку по разделителю KEGE и узнаем длины всех таких подстрок.

```
lens = list(map(len, s.split('KEGE')))
```

Для примера выше получим: [3, 4, 4, 3, 0]

Теперь нам нужно найти максимальную сумму трех подряд идущих слов и прибавить к ней 8 (2 строки *KEGE*) и 6 (*EGE* в начале и *KEG* в конце). Так как у первого слова слева и у последнего слова справа нет комбинаций *KEGE* уменьшим их длину на 3, чтобы наше рассуждение работало для всех последовательных троек.

```
lens[0] -= 3
lens[-1] -= 3
#  lens = [0, 4, 4, 3, -3]
#         [8 (0, 4, 4), 11 (4, 4, 3), 4 (4, 3, -3)] для примера
print(max(map(lambda a,b,c: sum(a,b,c), lens, lens[1:],
              lens[2:])) + 14)
```

Передача функции в качестве аргумента

Ранее в качестве аргумента при вызове, например функции *map* мы передавали известные нам функции (*len, max, sum*) или лямбда-выражения. На самом деле в качестве такого аргумента мы можем передавать и имена функций, которые написали сами.

Например, нам надо посчитать сколько налогов нужно будет платить с заработанной суммы в мифической стране X.

Мы знаем, что если человек заработал меньше 20000, то налог с него не удерживается, до 50000 – 4%, до 100000 – 6%, больше 100000 – 13%.

Да, мы можем написать лямбда выражение с вложенными тернарными операциями, но тогда код будет громоздким. Поэтому вынесем определение налога в отдельную функцию *ploti_nologi()*.

```
def ploti_nologi(money):
  if money < 20000:
    return 0
  elif money <= 50000:
    return money*0.04
  elif money < 100000:
    return money*0.06
  else:
    return money*0.13
```

Теперь, имея список с заработками граждан страны X, мы можем посчитать сумму налогов которые они заплатят.

```
profits = [15000, 54000, 22000, 143000, 20000]
print(sum(map(ploti_nologi, profits)))
```

Рекурсивная функция

Отдельная разновидность функций – рекурсивные функции. Рекурсивной функцией называется такая функция, которая в ходе своего исполнения вызывает сама себя, возможно, через другую функцию. Чтобы рекурсивная функция не вызывала сама себя бесконечно, определяют либо условие выхода из рекурсии, либо условие продолжения рекурсии.

Традиционный пример – вычисление факториала N!

Через условие продолжения рекурсии.

```
def factorial(n):
  if n > 0:
    return n*factorial(n-1)
  return 1
```

Через условие выхода из рекурсии.

```
def factorial(n):
  if n == 0:
    return 1
  return n*factorial(n-1)
```

Рекурсивные функции иногда весьма изящно описывают применяемый алгоритм, однако нужно проявлять некоторую аккуратность в их использовании.

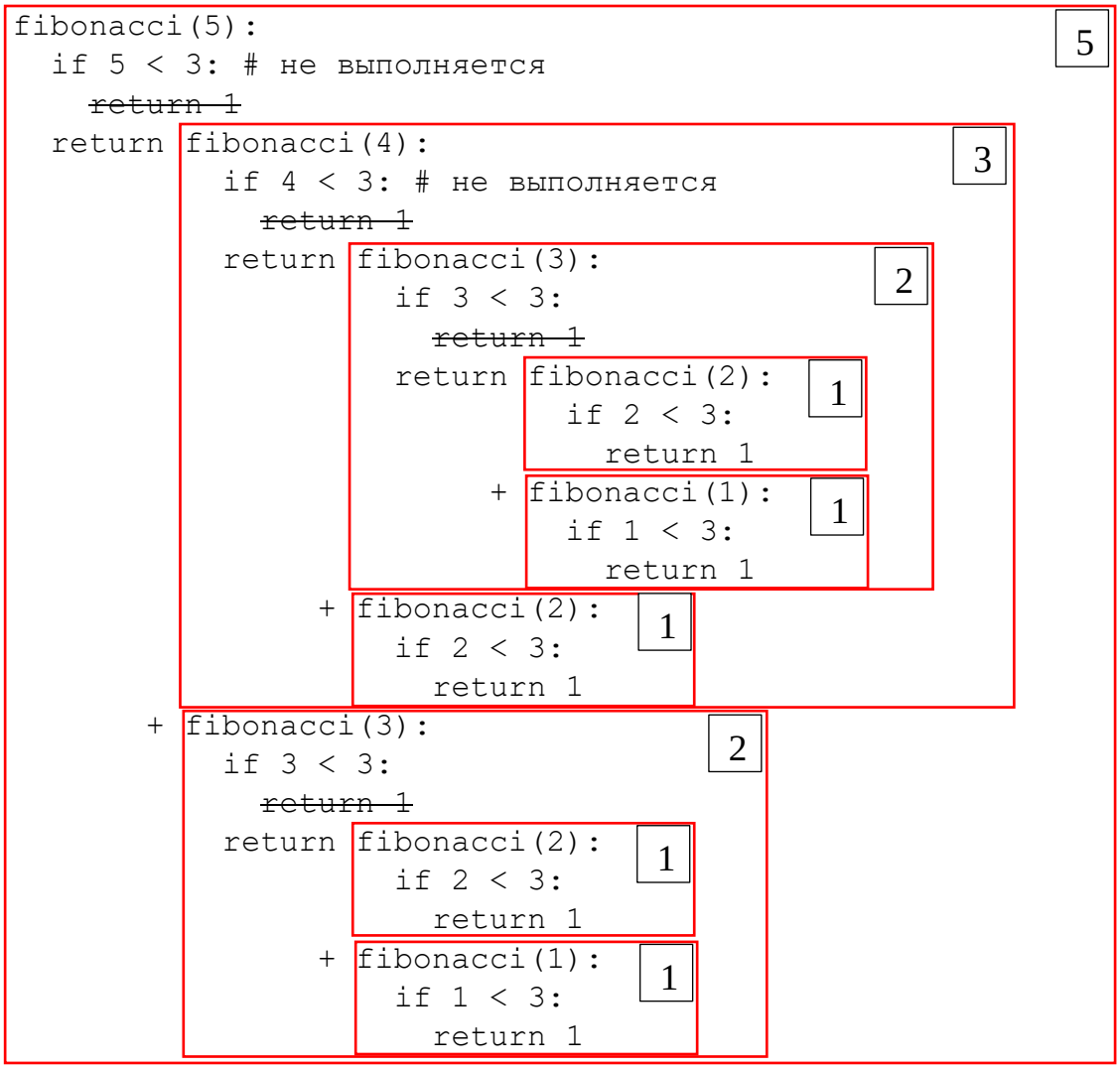
Определено это тем, что рекурсия работает вглубь и углубляется до тех пор, пока либо выполняется условие продолжения рекурсии, либо не будет соблюдаться условие выхода из рекурсии.

Рассмотрим эту особенность на еще одном традиционном примере – вычисление чисел Фибоначчи. Напомню, что числа Фибоначчи – это ряд чисел, начинающийся с двух единим и продолжающийся суммой предыдущих двух чисел – 1, 1, 2 (1+1), 3 (1+2), 5 (2+3), 8 (3+5), 13 (5+8), …

Рекурсивно *N*-число последовательности можно получить с помощью следующей функции на *Python*.

```
def fibonacci(N):
  if N < 3:
    return 1
  return fibonacci(N-1) + fibonacci(N-2)
```

Разберемся, как работает эта функция на примере вызова *fibonacci(5)*.



Оставим только последовательность вызовов функции:

```
fib(5) →
     (fib(4) →
          (fib(3) →
               (fib(2) → fib(1))
          ) →
          fib(2)
     ) →
     (fib(3) →
          (fib(2) → fib(1))
     )
```

Из последовательности можно сделать важные наблюдения.

Во-первых, каждое значение всегда считается заново. Во-вторых, любой вызов функции сначала доходит до точки выхода и только затем отдает управление обратно.

Второе наблюдение – вызов рекурсивной функции порождает цепочку вызовов этой же функции – стек вызовов. Это важно понимать, потому что глубина рекурсии в *Python* ограничена настройками среды. По умолчанию нельзя накапливать стек длиной больше 1000. Это можно избежать, если изменить системную переменную, ограничивающую глубину рекурсии. Делается это с помощью функции *setrecursionlimit* из библиотеки *sys*.

```
import sys
sys.setrecursionlimit(10000)
```

Однако, стоит помнить, что стоит избегать применения рекурсивных функций там, где их можно заменить более простым алгоритмом, например, циклическим. Или, как минимум, постараться реализовать рекурсивный алгоритм таким образом, чтобы для одних и тех же значений функция не вычисляла результат несколько раз.

Например, можно использовать следующую рекурсивную функцию, которая запоминает все вычисленные значения.

```
def fibonacci(N, fib_numbers=[1, 1]):
    if N <= len(fib_numbers):
        return fib_numbers[N-1]
    fib_numbers.append(fibonacci(N-1) + fibonacci(N-2))
    return fib_numbers[-1]
```

Данная функция имеет список вычисленных значений, к которому мы можем обратиться по имени параметра. Так как данный список инициализируется один раз при объявлении функции, изменяя его, мы можем расширять набор вычисленных значений. Также за счет природы выполнения рекурсивных

алгоритмов строка с добавлением нового элемента всегда будет добавлять первое невычисленное значение, перед тем, как добавить следующее.

Посмотрим, как изменится порядок вычисления. В скобках указан порядок возвращения значений рекурсивной функцией.

```
fib(5) → # выход из рекурсии для 5 (5)
     (fib(4) → # выход из рекурсии для 4 (3)
          (fib(3) → # выход из рекурсии для 3 (1)
               (fib(2) → fib(1)) # уже посчитано (0)
          ) →
          fib(2)# уже посчитано (2)
     ) →
     (fib(3) # это уже посчитано (4)
```

Можно придумать и другие способы хранения результатов работы рекурсивной функции. Однако в языке *Python* уже есть замечательный декоратор[1] *@lru_cache()* из библиотеки *functools*, который может сделать это за нас.

Теперь, если мы перепишем нашу первую реализацию рекурсивной функции в такую, которая не вычисляет значения для одного и того же числа *N* несколько раз, то получим следующий код.

```
from functools import lru_cache
@lru_cache()
def fibonacci(N):
  if N < 3:
    return 1
  return fibonacci(N-1) + fibonacci(N-2)
```

Код стал компактным, остался понятным и мы потратили на его оптимизацию крайне мало времени и сил.

При работе с декоратором *lru_cache* надо помнить, что все значения параметров должны быть строго неизменяемого типа. При попытке вызова функции с, например, списковым значением аргумента, *lru_cache* вернет ошибку «*TypeError: unhashable type: 'list'*» или «Ошибка типа: нехешируемый тип 'список'».

Также сохранение полученных результатов – ручное или с помощью *lru_cache* – не обходит проблему переполнения стека вызовов. Так как у нас каждому вызову функции *fibonacci()* будет предшествовать вызов функции *lru_cache()*, количество допустимых рекурсивных вызовов *fibonacci()* будет

[1] Декоратор – это функция, являющаяся своего рода «обёрткой» для другой функции. То есть такая функция, которая управляет запуском другой функции. Например, декоратор *@lru_cache()* перед запуском функции *fibonacci()* проверит, был ли уже запуск с таким же параметром и, если значение для такого параметра уже было найдено, сразу вернет результат без повторного запуска функции *fibonacci()*.
Подробнее работу с декораторами рассмотрим в будущих главах.

ограничено 500 (при каждом вызове в стек будет попадать сразу два вызова – для *lru_cache()* и для *fibonacci()*).

Однако, мы можем найти обходной путь – последовательно вычислить все значения до необходимого нам.

```
from functools import lru_cache
@lru_cache()
def fibonacci(N):
  if N < 3:
    return 1
  return fibonacci(N-1) + fibonacci(N-2)


list(map(fibonacci, range(1000)))
print(fibonacci(1000))
```

При таком подходе последовательный вызов функции *fibonacci()* позволит декоратору *lru_cache()* запомнить результаты для предыдущих значений (по умолчанию для последних 128) и наш алгоритм не будет совершать более двух рекурсивных вызовов для вычисления любого значения от 1 до 1000!

Модуль itertools

Полезные типы данных

Перед тем, как переходить к «ручному» написанию обработки последовательностей добавим в нашу копилку пару полезных типов данных.

Диапазон

Диапазон – генератор целых чисел в заданном диапазоне с указанным шагом.

Синтаксис объявления диапазона:

```
range(start, stop, step)
```

Генерирует последовательность от *start* до *stop* (не включая) с шагом *step*. В целом работает аналогично нахождению среза (генерирует такие же значения, как значения индексов в срезе).

Тип данных «диапазон» относится к последовательностям. Поэтому к переменным такого типа применимы все операции, допустимые для последовательностей, в том числе срезы и обращение по индексу.

```
>>> range(1, 100, 2)[10]
21
```

Однако, две операции над последовательностями – конкатенация (сложение) и повторение (дублирование) – диапазонами не поддерживаются.

Интересный факт.

Срез для диапазона возвращает диапазон, в котором начало, конец и шаг изменены в соответствии с указанными в срезе параметрами.

```
>>> range(10,100, 4)[10:5:-2]
range(50, 30, -8)
```

В данном примере первичный диапазон вернет числа 10, 14, 18, …, 90, 92, 96. Срез, соответственно, вернет числа 50, 46, 42, 38, 34 с шагом 2 или ряд 50, 42, 34. Как видим, шаг стал -8, начальное значение 50, первое не перечисляемое значение – 30 (из первичного среза без изменения шага).

Множество

Множеством называет набор уникальных значений. Множество является коллекцией, отличается коллекция от последовательности тем, что в ней нет строгого порядка следования элементов, поэтому нельзя обращаться к элементам множества по индексу.

Пустое множество объявляется так:

```
set_a = set()
```

Если необходимо заполнить множество уже известными значениями

```
set_a = {1, 2, 3, 'value'}
```

Также в множество можно добавить все уникальные значения итерируемого объекта, например, уникальные символы в строке или неповторяющиеся элементы списка.

```
uniq_str = set('Hello world!')  # {'H','e','l','o','w','r','d'}
uniq_list = set([1,2,3,2,3,4,1,5,2]) # [1, 2, 3, 4, 5]
```

Операции над множествами

Операция	**Обозначение**	**Пример**
Добавить элемент	`s.add(value)`	`s = set()` `s.add(5)  # {5}`
Объединение (все элементы множеств)	`A | B`	`>>>{1,2,3} | {2,3,4}` `{1,2,3,4}`
Пересечение (общие элементы множеств)	`A & B`	`>>>{1,2,3} & {2,3,4}` `{2,3}`
Вычитание (элементы A, которых нет в B)	`A - B`	`>>> {1,2,3} - {2,3,4}` `{1}`

Тождественность
(множества A и B содержат одни и те же элементы)
	`A == B`	`>>> {1,2} == {2, 1}`
		`True`
		`>>> {2,3,4} == {2,3}`
		`False`

Подмножество
(множество A входит в B)
	`A <= B`
	`A < B`
		`>>> {1,2,3}<={2,1,3,4}`
		`True`
		`>>> {1,2,3} < {3,1,2}`
		`False`

Супермножество
(множество A включает B)
	`A >= B`
	`A > B`
		`>>> {1,2,3} >= {2,1,3}`
		`True`
		`>>> {1,2,3} > {3,1,2}`
		`False`

Количество элементов	`len(s)`	`>>> len({1,3,5,1,7})` `4`

Также множества поддерживают операторы изменения объекта: `|=`, `&=`, `-=`.

Модуль itertools

На данный момент мы научились перебирать параллельные элементы в нескольких последовательностях с помощью *map*. Но что делать, если необходимо перебрать все возможные комбинации элементов последовательности?

На помощь приходит модуль *itertools*, который в числе прочего содержит функции, которые помогут последовательно перебрать такие комбинации.

Перед изучением следующих функций приведем общие свойства.

1) Итерируемые объекты полностью распаковываются перед обработкой.

То есть, если в качестве параметра передать итератор, сначала он будет распакован в последовательность и только затем обработан. Данное замечание очень важно, потому что результат бесконечных генераторов не может быть обработан таким образом. Также важный нюанс, что результат работы конечных итераторов может быть очень велик, что может вызвать проблемы, связанные с выделением памяти под объект-последовательность.

Например, при передаче файлового объекта в качестве параметра, то сначала весь файл будет прочитан и только потом обработана полученная последовательность.

2) Результаты возвращаются в возрастающем порядке. Порядок возрастания определяется исходя из обрабатываемой последовательности – меньше индекс элемента, меньше значение.

Например, если передать последовательность [3, 1, 11, 10], то возвращаемые комбинации будут получаться по правилу $3_1 < 1_2 < 11_3 < 10_4$. Нижние индексы отображают номер элемента в переданной последовательности.

Можно провести аналогию с цифрами в системах счисления – комбинация 1B54 будет точно идти после комбинации 01A2. Для примера из последовательности цепочка $11_3 1_2 11_3$ будет больше $3_1 1_2 11_3$.

3) Функции не проверяют элементы последовательность на одинаковость. Если при распаковке итерируемого объекта будут получены одинаковые элементы, то стоящий правее будет бóльшим.

Например, если передать последовательность [3, 3, 3, 3] с соответствующим порядком $3_1 < 3_2 < 3_3 < 3_3$. Функции будут воспринимать последовательности $3_1 3_1 3_2 3_3$ и $3_2 3_1 3_2 3_3$ как разны. Хотя с точки зрения значений элементов

последовательностей разницы нет (так как индекс не возвращается в сгенерированной последовательности).

4) Результат вызова любой рассматриваемой функции – итератор.

Генерируемая последовательность не возвращается сразу вся и, соответственно, не хранится вся в памяти. Поэтому, если необходимо получить последовательность сгенерированных последовательностей, нужно привести итератор к списку или кортежу.

Размещения без повторений

Функция, возвращающая все размещения длиной *r* элементов последовательности – *itertools.permutations*.

```
itertools.permutations(итерируемый_объект, r=длина_размещения)
```

Предположим, что у нас имеет последовательность *s* из 3 элементов – *[s1, s2, s3]*. Результатом работы функции `permutations(s, r=3)` будет итератор, который вернет следующие кортежи всегда в таком порядке, независимо от того, есть ли в последовательности повторяющиеся элементы:

```
(s1,s2,s3),(s1,s3,s2),(s2,s1,s3),(s2,s3,s1),(s3,s1,s2),(s3,s2,s1)
```

Примеры.

```
>>> from itertools import permutations
>>> list(permutations('abc', r=3))
[('a', 'b', 'c'), ('a', 'c', 'b'), ('b', 'a', 'c'),
('b', 'c', 'a'), ('c', 'a', 'b'), ('c', 'b', 'a') ]

>>> list(permutations('cccc', r=2))
[('c','c'),('c','c'),('c','c'),('c','c'),('c','c'),('c','c'),
('c','c'),('c','c'),('c','c'),('c','c'),('c','c'),('c','c')]
```

Что? Почему 12 одинаковых перестановок? Дело в том, что *permutations* отличает все символы *c* (в такой постановке у нас 4 разных символа *c*) и выдает нам перестановки каждого символа *c* со всеми остальными (4·3 штук).

Размещения с повторениями

Для работы с размещениями с повторениями есть функция *product*.

Функция *product* возвращает размещениями с повторениями для элементов передаваемых итерируемых объектов длиной, указанной в параметре *repeat*. Если значение *repeat* не задано, то для нескольких итерируемых объектов комбинации, где на первом месте будут расположены элементы первого итерируемого объекта, на втором – второго и т.д.. Если же в качестве аргументов был подан только один итерируемый объект, то в качестве результата последовательно вернуться кортежи из одного элемента.

Для последовательности *s* из 3 элементов – *[s1, s2, s3]*. Результатом работы функции `product(s, repeat=2)` будет итератор, который вернет следующие кортежи всегда в таком порядке, независимо от того, есть ли в последовательности повторяющиеся элементы:

```
(s1,s1),(s1,s2),(s1,s3),(s2,s1),(s2,s2),(s2,s3),(s3,s1),(s3,s2),
(s3,s3)
```

Примеры.

```
>>> from itertools import product
>>> list(product('123'))
[('1',), ('2',), ('3',)]
>>> list(product('12', repeat=2))
[('1','1'), ('1','2'), ('2','1'), ('2','2')]


>>> list(product('abc', 'xy'))
[('a','x'), ('a','y'), ('b','x'), ('b','y'), ('c','x'), ('c','y')]


>>> list(product('bb', 'yyz'))
[('b','y'), ('b','y'), ('b','z'), ('b','y'), ('b','y'), ('b','z')]

>>> list(product('01', 'xy', repeat=2))
[('0', 'x', '0', 'x'), ('0', 'x', '0', 'y'), ('0', 'x', '1', 'x'),
('0', 'x', '1', 'y'), ('0', 'y', '0', 'x'), ('0', 'y', '0', 'y'),
('0', 'y', '1', 'x'), ('0', 'y', '1', 'y'), ('1', 'x', '0', 'x'),
('1', 'x', '0', 'y'), ('1', 'x', '1', 'x'), ('1', 'x', '1', 'y'),
('1', 'y', '0', 'x'), ('1', 'y', '0', 'y'), ('1', 'y', '1', 'x'),
('1', 'y', '1', 'y')]
```

Работу последнего примера можно проиллюстрировать с помощью следующего кода

```
>>> list(product(product('01', 'xy'), repeat=2))
[(('0', 'x'), ('0', 'x')), (('0', 'x'), ('0', 'y')), (('0', 'x'),
('1', 'x')), (('0', 'x'), ('1', 'y')), (('0', 'y'), ('0', 'x')),
(('0', 'y'), ('0', 'y')), (('0', 'y'), ('1', 'x')), (('0', 'y'),
('1', 'y')), (('1', 'x'), ('0', 'x')), (('1', 'x'), ('0', 'y')),
(('1', 'x'), ('1', 'x')), (('1', 'x'), ('1', 'y')), (('1', 'y'),
('0', 'x')), (('1', 'y'), ('0', 'y')), (('1', 'y'), ('1', 'x')),
(('1', 'y'), ('1', 'y'))]
```

Заметим, что в таком случае имеем дело с парами кортежей из *seq*. В предыдущем примере этап формирования кортежей пропущен, поэтому имеем дело с четверками, порядок следования элементов в которых соответствует порядку следования в парах кортежей.

Например, вместо `(('0', 'x'), ('1', 'y'))`
имеем `('0', 'x', '1', 'y')`).

Сочетания

Функция для получения сочетаний без повторений – *itertools.combinations*. Количество элементов в сочетании устанавливается через параметр *r*, который является обязательным параметром.

Для последовательности *s* из 3 элементов – *[s1, s2, s3]*. Результатом работы функции `combinations(s, r=2)` будет итератор, который вернет следующие кортежи всегда в таком порядке, независимо от того, есть ли в последовательности повторяющиеся элементы:

```
(s1,s2), (s1,s3), (s2,s3)
```

Примеры.

```
>>> from itertools import combinations
>>> list(combinations('bbb', r=2))
[('b', 'b'), ('b', 'b'), ('b', 'b')]
```

```
>>> from itertools import combinations
>>> list(combinations('abc', r=2))
[('a', 'b'), ('a', 'c'), ('b', 'c')]
```

Про уникальные комбинации

Если же мы хотим обработать только уникальные комбинации или не обрабатывать одинаковые, можно преобразовать возвращаемый итератор в множество и обработать только уникальные значения.

```
>>>from itertools import combinations
>>>set(combinations('bbc', r=2))
{('b', 'b'), ('b', 'c')}
```

Функция zip

Функция запаковки *zip* – возвращает итератор кортежей, где *i*-тый кортеж содержит *i*-е элементы всех переданных аргументов.

Говоря иначе, итератор кортежей возвращает кортежи вида (a_i, b_i, …, z_i), где *a*, *b*, …, *z* – итерируемые объекты, переданные в качестве аргументов функции *zip*, *i* – допустимый для всех объектов индекс.

Кортежи возвращаются по возрастанию индекса *i* до тех пор, пока на возвращаемой позиции всех переданных последовательностей есть значение.

Рассмотрим пример с запаковкой трех последовательностей (см.рис.2).

```
iter1 = [8, -3, 1, 0, 13, 7, 19]
iter2 = [9, 5, 4, -6]
iter3 = [3, 1, 5, 1, 71]
```

Тогда итератор z

```
z = zip(iter1, iter2, iter3)
```

Будет последовательно возвращать кортежи (8, 9, 3), (-3, 5, 1), (1, 4, 5), (0, -6, 1).

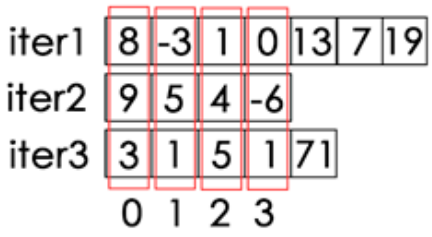


Рисунок 2. Пример запаковки

Циклы for и while. Простые алгоритмы

Цикл for

Конструкция *for* в языке *python* значительно отличается от подобной конструкции в других языках. В *python* данная конструкция служит только для перебора элементов итерируемого объекта.

Синтаксически конструкция выглядит так:

```
for переменные in итерируемый_объект:
  код_для_обработки
```

Говоря иначе, каждый элемент итерируемого объекта распаковывается в список переменных или сохраняется в одной переменной.

Одно выполнение списка команд, записанных внутри цикла, называется итерацией.

Важно понимать, что все операции, описанные внутри цикла, выполняются от первой строки до последней (кроме случаев, когда выполнение какой-либо команды приводит к ошибке или когда указаны специальные команды, которые управляют работой цикла).

Примеры.

```
>>>for x in [1, 3, 7, 13]:
>>>  print(x)
1
3
7
13

>>>for i in range(2, 10, 2):
>>>  print(i, end = ':')
2:4:6:8:


>>>from itertools import product
>>>l = []
>>>for a, b in product([2,4,6], [3,4,5]):
>>>  if (a + b) % 2 == 0:
>>>    l.append(a+b)
>>>l
[6, 8, 10]
```

Операторы continue и break

Оператор *continue* – переход на следующую итерацию цикла без выполнения кода текущей итерации до конца.

Оператор *break* – выход из цикла без завершения кода текущей итерации и выполнения последующих за строкой с *break* команд.

При этом итерация считается выполненной, то есть прерывание выполнения тела цикла именно завершает итерацию без выполнения последующих команд.

Примеры.

```
>>>for a in [1,3,5,7,9,11]:
>>>  if a > 8:
>>>    break
>>>  print(a*2, end=' ')
2, 6, 10, 14
```

```
>>>for a in [1,3,5,7,9,11]:
>>>  if 4 <= a < 8:
>>>    continue
>>>  print(a*2, end=' ')
2, 6, 18, 22
```

Конструкция for…else

Также конструкция *for..in* в python имеет возможность проверить, завершился ли цикл без выхода с помощью оператора *break*.

```
for переменные in итерируемый_объект:
  тело цикла
else:
  если цикл окончил работу без остановки с помощью break
```

Пример.

```
>>>for i in [2, 10, 20, 50, 4, 70]:
>>>  if i <= 50:
>>>    print('Не все числа больше, чем 50')
>>>    break
>>>else:
>>>  print('Все числа больше, чем 50')
```

Примеры использования

Перебрать все целочисленные концы диапазонов между целыми значениями a и b.

```
from itertools import combinations
a, b = map(int, input().split())
print(list(combinations(range(a, b+1), r=2)))
```

Пример вывода для $a = 2$ и $b = 5$.

```
[(2, 3), (2, 4), (2, 5), (3, 4), (3, 5), (4, 5)]
```

Построить таблицу истинности для выражения $a \wedge b \rightarrow \neg c$

```
from itertools import product
for a, b, c in product([0, 1], repeat=3):
  print(a, b, c, (a and b) <= (not c))
```

Вывод

```
0 0 0 True
0 0 1 True
0 1 0 True
0 1 1 True
1 0 0 True
1 0 1 True
1 1 0 True
1 1 1 False
```

Сколько последовательностей длиной 5 можно составить путем перестановки букв слова ПРИМЕР?

```
from itertools import permutations
print(len(set(permutations('ПРИМЕР'))))
```

Сколько в списке смежных пар таких, что их сумма кратна 10? Смежной парой называется пара подряд идущих чисел.

```
nums = [1, 5, 3, 7, 5, 6, 4 ,6, 6 ,2, 8]
res = []
for a, b in zip(nums, nums[1: ]):
  if (a + b) % 10 == 0:
    res.append(a+b)
print(len(res))
```

Цикл while

Также бывают случаи, когда итерации необходимо выполнять до тех пор, пока соблюдается какое-то условие. И именно для решения таких задач существует конструкция цикла *while*.

```
while условие_продолжения:
  # тело цикла
```

Тело цикла будет повторно выполняться до тех пор, пока истинно условие.

Сделаем трассировку простого алгоритма, использующего цикл *while*.

```
1. x = 5
2. while x < 30:
3.   x += 7
4. print(x)
```

Номер строки	**Команда**	**x**	**x < 30**
1	`x = 5`	5	
2	`while x < 30:`		5 < 30 # True
3	`    x += 7`	12	
2	`while x < 30:`		12 < 30 # True
3	`    x += 7`	19	
2	`while x < 30:`		19 < 30 # True
3	`    x += 7`	26	
2	`while x < 30:`		26 < 30 # True
3	`    x += 7`	33	
2	`while x < 30:`		33 < 30 # False
4	`print(x)`		

Как можно заметить, цикл выполнил 4 итерации и завершился, когда значение *x* перестало соответствовать условию *x < 30*.

Пример.

С клавиатуры вводятся числа. Необходимо накопить сумму введенных чисел, большую, чем 100. Вывести на экран первую такую сумму.

```
s = []
while sum(s) <= 100:
  x = int(input())
  s.append(x)
print('Sum = ', sum(s))
```

Операторы continue и break

Как и в случае с циклом *for* для цикла *while* существует пара операторов, которая позволяет управлять выполнением циклического алгоритма.

Оператор *continue* – пропускает весь код до конца тела цикла и переходит к следующей итерации.

Оператор *break* – прерывает выполнение цикла на строке, в которой вызван.

Конструкция while…else

Данная конструкция аналогична конструкции *for…else*, код, написанный в блоке *else*, выполняется только в том случае, если цикл был окончен без использования оператора *break*.

Простые алгоритмы

Рассмотрим реализацию простых алгоритмов, на основе которых в дальнейшем будем строить более сложные алгоритмы.

Реализация счетчика

Идея алгоритма очень простая – если при последовательном переборе вариантов встречаем удовлетворяющее условию значение, увеличиваем счетчик на 1. Перед началом перебора принимаем его значение за 0.

Псевдокод алгоритма:

```
count = 0
цикл_для_перебора:
  if условие_отбора:
    count += 1
```

Пример.

Сколько чисел, входящих в диапазон [1000;5000], кратны 3 и не кратны 5?

```
count = 0
for x in range(1000; 5001):
  if x % 3 == 0 and x % 5 != 0:
    count += 1
print(count)
```

Нахождение суммы/произведения

Идея алгоритма аналогична алгоритму для счётчика, за тем исключением, что мы добавляем не единицу, а найденное число. Для нахождения суммы изначально задаем значение 0 ($0+x=x$), для произведения – 1 ($1 \cdot x = x$).

Псевдокод алгоритма для нахождения суммы:

```
s = 0
цикл_для_перебора:
  if условие_отбора:
    s += найденное_значение
```

Псевдокод алгоритма для нахождения произведения:

```
p = 1
цикл_для_перебора:
  if условие_отбора:
    p *= найденное_значение
```

Пример.

В многострочном файле *test.txt* посчитайте суммарное количество букв *A* в строках, длина которых больше 100.

```
f = open('test.txt')
line = f.readline()
s = 0
while line:
  if len(line) > 100:
    s += line.count('A')
  line = f.readline()
print(s)
f.close()
```

Нахождение минимума/максимума

Логика этого алгоритма строится на следующем рассуждении: если текущее подходящее значение меньше (для нахождения минимума) уже найденного значения минимума, тогда переопределяем данное значение.

Аналогичное рассуждение производится для максимума. В качестве начальных значений на языке *Python* удобно использовать специальные числовые объекты – значения бесконечностей – *float('-inf')* и *float('inf')*.

Обобщенный алгоритм для минимума:

```
m = float('inf')
цикл_для_перебора:
  if условие_отбора and m > найденное_значение:
    m = найденное_значение
```

Для нахождения максимума достаточно изменить «плюс бесконечность» на «минус бесконечность» и знак сравнения.

Однако, в отличии от предыдущих алгоритмов, у алгоритмов нахождения максимума и минимума может быть требование к нахождению первого или последнего совпадения.

Предложенный ранее обобщенный алгоритм находит первое минимальное значение, так как ищется значение строго меньшее. Если же нам необходимо найти последнее совпадение – нужно заменить строгий знак на нестрогий.

Обобщенный алгоритм для нахождения последнего минимального значения:

```
m = float('inf')
цикл_для_перебора:
  if условие_отбора and m >= найденное_значение:
    m = найденное_значение
```

Основная идея динамического программирования

Стоит отметить, что рассмотренные алгоритмы реализации счётчика, нахождения суммы/произведения/минимума/максимума являются самыми простыми примерами динамического программирования.

Идея динамического программирования следующая: нет необходимости сохранять весь обработанный массив данных, достаточно сохранить только те значения, которые будут полезны по отношению к решаемой задаче.

Например, мы ищем четный максимум и уже обработали *N* элементов.

Рисунок 1. Пример обрабатываемой последовательности

Что нам нужно знать об уже обработанных N элементах?

Во-первых, были ли вообще удовлетворяющие условию элементы. Для этого принято использовать «нейтральные» значения – такие значения, которые не удовлетворяют ограничениям задачи. Например, любое нечетное значение или значение для минус бесконечности (*float('-inf')*).

Определив такие значения, после выполнения алгоритма мы можем понять, нашли искомое значение или нет. Ведь в последовательности может и не быть элементов, удовлетворяющих условию поиска. Для рассматриваемой задачи – последовательность из нечетных элементов не содержит нужных значений.

Во-вторых, указать условие, которое определит найден ли подходящий элемент. В случае с четным максимумом это признак четности (x % 2 == 0).

В-третьих, указать условие для нахождения нового максимума.

Подытожим и перенесем наше рассуждение в псеводкод на *Python*.

Для начального нечетного

```
mx = 3
цикл_для_перебора:
  x = вычисляем_значение
  if x%2==0 and (mx==3 or mx<x):
    mx = x
```

Для начального бесконечного

```
mx = float('-inf')
цикл_для_перебора:
  x = вычисляем_значение
  if x%2==0 and \
   (mx==float('-inf') or mx<x):
    mx = x
```

При работе данного алгоритма первое найденное чётное значение сохраняется, так как проходит проверка на нейтральное значение. После чего первое условие в скобке перестанет выполняться и алгоритм будет постепенно

находить максимальное чётное значение. Если после завершения алгоритма значение переменной *mx* останется начальным, значит в последовательности не было чётных чисел.

В чем же существенное отличие значений *3* и *float('-inf')*? Все дело в том, что значение *float('-inf')* меньше любого значения, которое может встретиться при переборе последовательности. Поэтому алгоритм можно преобразовать следующим образом.

Начальная версия

```
mx = float('-inf')
цикл_для_перебора:
  x = вычисляем_значение
  if x%2==0 and \
   (mx==float('-inf') or mx<x):
    mx = x
```

Оптимизированная версия

```
mx = float('-inf')
цикл_для_перебора:
  x = вычисляем_значение
  if x % 2 == 0 and mx < x:
    mx = x
```

Тогда, если начальное значение переменной *mx* останется равным *float('-inf')*, значит в последовательности не было удовлетворяющих элементов.

Преимущество таких алгоритмов заключается в эффективном использовании памяти и скорости работы. Потому что нет необходимости запускать встроенные функции для всего обрабатываемого потока данных и формировать новые «тяжелые» структуры в памяти.

Пример реализации на встроенных функциях

```
nums = map(int, open('file'))
evens = filter(lambda x: x % 2 == 0, nums)
print(max(evens))
```

Динамический пример

```
mx = float('-inf')
for x in map(int, open('file'):
  if x % 2 == 0 and mx < x:
    mx = x
print(mx)
```

В некоторых случаях подобные алгоритмы позволяют преобразовать переборы с нелинейной сложностью (переборы пар, троек и т.п) в линейные.

Например, если необходимо найти количество пар чисел с четной суммой в списке целых чисел *nums*.

```
from random import randrange
nums = [randrange(-10000, 10000) for _ in range(10000)]
```

Переборный алгоритмы

```
from itertools import combinations

count = 0
for t in combinations(nums, r=2):
  if sum(t) % 2 == 0:
    count += 1
print(count)
```

Однако мы можем подумать о динамическом программировании и понять, что четные суммы образуют пары чисел с одинаковой четностью. Поэтому достаточно посчитать количества четных и нечетных значений и по формуле сочетаний вычислить интересующее нас количество.

Статистический алгоритм

```
odd, even = 0, 0
for x in nums:
  if x % 2 == 0:
    even += 1
  else:
    odd += 1
print((even*(even-1) + odd*(odd-1)) // 2)
```

Или можем считать количество четных сумм на каждой итерации, использовав подход динамического программирования.

Динамическое программирование

```
count, odd, even = 0, 0, 0
for x in nums:
  if x % 2 == 0:
    count += even
    even += 1
  else:
    count += odd
    odd += 1
print(count)
```

Скорость работы переборного алгоритма можно наглядно сопоставить со скоростью работы последних двух алгоритмов на примере

последовательности из 10 тысяч чисел. Так переборный алгоритм будет считать порядка 10 секунд, в то время как другим алгоритмам для этого понадобятся доли секунды.

Реализации более сложных алгоритмов изучим в следующих главах.

Перебор цифр числа

Еще одним популярным алгоритмом обработки чисел является алгоритм перебора разрядов числа.

Ручной алгоритм перевода выглядит следующий образом.

Число делится на основание системы счисления – остаток от деления является младшим (правым) разрядом, полученный результат деления (целая часть) делится на основание системы счисления – остаток от деления второй справа разряд. Данные действия повторятся до тех пор, пока результат деления не станет равен нулю.

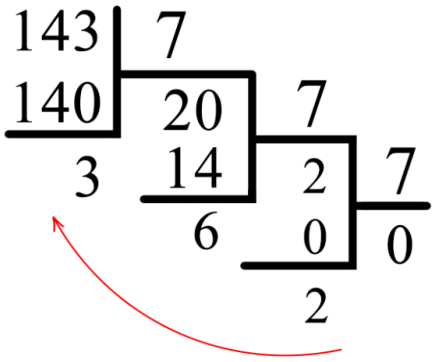


Рисунок 2. Пример перевода 143_{10} в семеричную систему счисления

Таким образом мы можем заключить, что алгоритм последовательного деления до нуля позволяет перебрать все разряды числа в *n*-ричной системе счисления.

Как же будет выглядеть такой алгоритм на языке *Python*?

```
x = int(input('Введите число в 10 сс:'))
n = int(input('Введите основание конечной сс:'))
# пока не дошли до нуля
while x > 0:
  # определяем младший ненайденный разряд
  digit = x % n
  # находим число для следующего шага
  x = x // n
```

Данный алгоритм можно расширять с помощью рассмотренных ранее базовых алгоритмов. Например, для подсчета количества разрядов с определенным значением или суммы всех разрядов.

Генераторы. Функции all и any

В данной главе мы рассмотрим как работают генераторы на самом общем уровне – узнаем, что это такое и какой принцип формирования значений используется при работе. Более глубокий анализ и продвинутые методы работы с генераторами в *Python* будут освещены в одной из следующих глав.

Генераторы

Генераторы, как можно понять из названия типа объектов, это объекты, которые генерируют значения. Говоря иначе при каждом новом запросе вычисляют новое значение по заданному алгоритму.

Генераторы ведут себя как итераторы, то есть вычисляют значения по мере обращения к ним. То есть при многократном вызове метода *next(gen)*, где *gen* – созданный генератор, мы последовательно получаем генерируемую последовательность.

Генератор не является последовательностью, то есть нельзя обратиться к возвращаемым элементам генератора по индексу.

Генераторы реализуют модель ленивых вычислений. Говоря иначе, следующее значение вычисляется только тогда, когда в программе указана команда возвращения этого значения. Например, при очередном вызове функции *next*.

Выражения-генераторы

Выражения-генераторы очень удобный инструмент для создания итераторов.

Синтаксически описание выражения-генератора выглядит так:

```
выражение_генератора ::=
(выражение for набор_переменных1 in итер.об1 [if условие1]
          [for набор_переменныхN in итер.обN [if условиеN]])
```

В итоге возвращается генератор который выдает значения выражения, которые генерируются в соответствии с порядком описанным в циклах *for* и соответствующие указанным условиям.

Стоит отметить, что переменные в таких генераторах определяются слева направо и *условиеK* может содержать любую из переменных, входящих в наборы *1..K*.

Пример 1.

Генератор квадратов чисел из списка.

```
a = [5, 2, 10, 9]
gen = (x*x for x in a)
print(list(gen))
# [25, 4, 100, 81]
```

Данный алгоритм выводит на экран значения, аналогичные следующему алгоритму.

```
gen_list = []
for x in a:
  gen_list.append(x*x)
print(gen_list)
```

Пример 2.

Генератор целых значений по модулю, считанных из многострочного файла *test.txt*.

```
gen = (int(s) if s[0]!='-' else -int(s) for s in open('test.txt'))
print(list(gen))
```

Пояснение: из файла последовательно считываются строки. Если строка не начинается на '-', то возвращаем преобразованную в число строку, в противном случае, возвращаем число, переведенное из строки, с обратным знаком. Или для каждого значения *s* вычисляется выражение, записанное с помощью тернарного оператора, `int(s) if s[0]!='-' else -int(s)`.

Алгоритм с аналогичным результатом:

```
gen_list = []
for s in open('test.txt'):
  if s[0] != '-':
    gen_list.append(int(s))
  else:
    gen_list.append(-int(s))
print(gen_list)
```

Пример 3.

Генератор сумм пар чисел в диапазоне 0..5, в которых хотя бы один элемент кратен 3.

```
gen = (a + b for a in range(5)
             for b in range(5) if a % 3 == 0 or b % 3 == 0)
print(list(gen))
# (0, 1, 2, 3, 4, 1, 4, 2, 5, 3, 4, 5, 6, 7, 4, 7)
```

Алгоритм с аналогичным результатом:

```
gen_list = []
for a in range(5):
  for b in range(5):
    if a % 3 == 0 or b % 3 == 0:
      gen_list.append(a + b)
print(gen_list)
```

Генератор для пар чисел, где оба числа должны быть кратны трем:

```
gen = (a + b for a in range(5) if a % 3 == 0
             for b in range(5) if b % 3 == 0)
print(list(gen))
```

Алгоритм с аналогичным результатом:

```
gen_list = []
for a in range(5):
  if a % 3 == 0:
    for b in range(5):
      if b % 3 == 0:
        gen_list.append(a + b)
print(gen_list)
```

В данном примере важно заметить одну деталь. Дело в том, что при предварительной проверке на кратность трем значения переменной *a* алгоритм не будет перебирать значения *b*, если условие будет ложным. Такой подход позволяет перебирать значения быстрее, не инициализируя вложенный цикл без необходимости.

Пример 4.

Генератор для проверки соответствия наборов значений переменных условию $(a \wedge \neg b \vee c)$.

```
from itertools import product
gen = (a and not(b) or c for a, b, c in product([0, 1], repeat=3))
print(list(gen))
# (0, 1, 0, 1, True, True, 0, 1)
```

Но почему получились разные результаты?

Все дело в порядке вычисления. Дело в том, что в случаях, когда имеем случай $a \neq 1$ или $b \neq 0$, результат выражения зависит от значения c. Поэтому возвращается значения c. Если же $a = 1$ и $b = 0$, то значение *a and not (b)* равно True, так как является результатом выполнения логической операции *and*.

Алгоритм с аналогичным результатом:

```
from itertools import product
gen_list = []
for a, b, c in product([0, 1], repeat=3):
  gen_list.append(a and not(b) or c)
print(gen_list)
```

List comprehension и set comprehension

Как можно заметить, генераторы синтаксически объявляются в круглых скобках. При этом такая запись не формирует кортеж, а именно объект генератора-выражения.

В *python* существует две формы записи для быстрого преобразования возвращаемых генератором значений в список и множество.

При применении вместо круглых скобок квадратных последовательность будет сразу распакована в список (*list comprehension*, также списковое включение). Такой же приём можно применить и для преобразования в множество (*set comprehension*).

```
[выражение_генератора]  # список значений
{выражение_генератора}  # множество значений
```

List comprehension и *set comprehension* помимо удобства использования помогают использовать вычислительные ресурсы эффективнее, чем привычное преобразование через приведение к типу через *list(gen)* и *set(gen)*.

К сожалению, подобного механизма для кортежей нет, в качестве альтернативы *tuple(gen)* можно использовать запись через распаковку с помощью операторов астерикса и запятой (без запятой запись не работает).

```
*(выражение_генератор),
```

Такая запись соответствует *singleton tuple* нотации и запаковывает две последовательности (результат работы *gen* и пустое значение) в кортеж.

Стоит отметить, что такая запись не имеет преимуществ перед приведением через функцию *tuple(gen)* в плане потребления вычислительных ресурсов.

Подобный принцип также можно использовать и при присвоении значений работы генератора в переменную в виде списка.

```
*переменная_списка, = (выражение_генератор)
```

Однако, предпочтительным вариантом сохранения списка, являющимся результатом работы выражения-генератора, является *list comprehension*.

Функции-генераторы

Функции-генераторы – особая разновидность функций, которые ведут себя как итераторы. То есть, при объявлении такой функции создается объект генератор, к которому можно обратиться с помощью функции *next*.

Осуществляется это с помощью ключевого слова *yield*.

Пример для иллюстрации.

Создадим функцию-генератор *test_gen*, которая будет возвращать 3 значения по порядку, например, 5, 2 и 7.

```
>>>def test_gen():
  print('Generate first value')
  yield 5
  print('Generate second value')
  yield 2
  print('Generate third value')
  yield 7
```

Для обращения к генератору-функции нужно создать объект генератора функции (такая тавтология).

```
>>>gen_obj = test_gen()
```

Что из себя представляет объект и как работает такая функция?

При вызове функции возвращается объект «генератор-функция». Выполнение такой функции «заморожено» в самом начале. Условно можно изобразить такой объект следующим образом, где вертикальная красная черта (далее курсор) определяет место текущего выполнения функции.

```
gen_obj =
 |print('Generate first value')
  yield 5
  print('Generate second value')
  yield 2
  print('Generate third value')
  yield 7
```

После выполнения функции *next(gen_obj)* выполняются все команды после курсора, включая ближайший вызов оператора *yield*.

```
>>>gen_obj(next)
'Generate first value'
5
```

При этом объект генератора функции зафиксирует курсор в позиции после первого оператора *yield*.

```
gen_obj =
  print('Generate first value')
  yield 5
 |print('Generate second value')
  yield 2
  print('Generate third value')
  yield 7
```

То есть мы видим, что код функции не выполняется до тех пор, пока мы не указываем, что нам необходимо получить следующее значение.

```
>>>next(gen_obj)
'Generate second value'
2
>>>next(gen_obj)
'Generate third value'
7
```

Но что будет после того, как генератор вернет все значения? Сейчас наша генератор-функция находится в следующем состоянии.

```
gen_obj =
  print('Generate first value')
  yield 5
  print('Generate second value')
  yield 2
  print('Generate third value')
  yield 7|
```

При последующем вызове функции *next(gen_obj) python* будет возвращать код исключения *StopIteration*.

Следующая конструкция выведет на экран все возвращаемые генератором значения

```
for a in test_gen():
  print(a)
```

Если представить данный код с помощью цикла *while* и обработки исключения, получим такую запись.

```
gen = test_gen()
try:
  while True:
    a = next(gen)
    print(a)
except StopIteration:
  print('Stop iteration')
```

Конструкция *try-except* выполняет код, описанный в секции *try*, и в случае возникновения исключения *StopIteration* печатает на экран фразу *'Stop Iteration'*.

Конечно же писать подобные генераторы через функции не самый грамотный вариант решения задачи, особенно учитывая наличие выражений-генераторов.

Функции-генераторы применяются для решения задач, где нельзя обойтись простыми генераторами-выражениями.

Например, считывание строк файла, который дозаписывается в процессе его чтения.

```
import time

def file_read(filename):
  f = open(filename)
  s = ''
  while s != 'STOP':
    s = f.readline()
    if not s:
      time.sleep(0.1)
      continue
    yield s
  f.close()

for line in file_read('test.txt'):
  print(line.strip())
```

Зачем нужен такой генератор? Все просто, им очень удобно пользоваться. Да, можно написать подобный код и без генератора, но тогда насколько менее читаемым будет код, где мы совместим алгоритм считывания с ожиданием и обработкой строки. В рассмотренном же случае мы пишем логику считывания файла внутри функции-генератора и используем его внутри цикла *for*, что разделяет логические части нашего алгоритма.

Функции all и any

Очень удобным инструментом совместно с применением генераторов являются функции *all* и *any*.

Функция *all(итер_объект)* проверяет, все ли значения итерируемого объекта истинны, функция *any(итер_объект)* – истинен ли хотя бы один элемент итерируемого объекта.

Функция *all* работает по следующей логике: перебираются элементы итерируемого объекта, если найдено значение, соответствующее *False* (0, пустая строка, пустой список, *None*), то возвращается *False*, если таких элементов не найдено возвращается *True*.

Работу функции *all* можно проиллюстрировать с помощью следующего кода:

```
def all_func(iter):
  for x in iter:
    if not x:
      return False
  return True
```

Функция *any* работает так: если при переборе встретилось значение, соответствующее значению *True*, то возвращается *True*, если таких значений не встретилось, возвращается *False*.

```
def any_func(iter):
  for x in iter:
    if x:
      return True
  return False
```

Также можно использовать данные функции с оператором *not*.

not any(итер_объект) – все значения ложны,

not all(итер_объект) – хотя бы одно значение ложно.

Пример 1.

Все ли целые числа из диапазона [10; 40] удовлетворяют выражению?

$$x^2 + 3x - 10 > 0$$

Решение с помощью *all* (выводит *True* или *False*)

```
print(all(x**2 + 3x - 10 > 0 for x in range(10, 41)))
```

Пример 2.

Все ли целые числа, кратные 3 или 5, в диапазоне [100; 150] имеют еще один нечетный делитель, отличный от 1, 3 и 5?

Решение.

Разложим задачу на подзадачи.

Определение кратности 3 или 5

```
any(x % d == 0 for d in (3, 5))
```

Определение наличия нечетного делителя – переберем все числа от 7 до x с шагом два, если хотя бы одно число является делителем, вернем *True*.

```
any(x % d == 0 for d in range(7, x, 2))
```

Соединяем в одно выражение

```
print(all(any(x % d == 0 for d in range(7, x, 2))
          for x in range(100, 151)
          if any(x % d == 0 for d in (3, 5))))
```

Полезные приемы

Сумма логических значений

Также с помощью выражений-генераторов и понимания, что логический тип наследуется от целочисленного, можно находить количество совпадений с помощью функции *sum*.

Пример.

Сколько чисел в диапазоне [1000; 10000] делятся на 3, 5 и 7 одновременно?

Решение.

```
sum(all(x % d == 0 for d in (3, 5, 7)) for x in range(1000, 10001))
```

В итоге получим последовательность из значений *True* и *False*, которые при суммировании будут интерпретироваться, как 1 и 0, соответственно.

Аналогично можно воспользоваться функцией нахождения длины для конструкции с *list comprehension*

```
len( [0 for x in range(1000, 10001)
        if all(x % d == 0 for d in (3, 5, 7))] )
```

Так как нет необходимости сохранять все значения для скорости работы можно присвоить всем элементам одно значение, которое уже хранится в памяти.

Многомерные списки

Также стоит помнить, что результат работы генератора на каждом этапе возвращает ту структуру, которая описана в левой части выражения. То есть, если указывается кортеж, то генерироваться будет последовательностей кортежей.

Пример.

Составить список значений, который можно получить из чисел 2, 5, 8, 10, 15 путем применения двух команд +2 и *3.

Реализация 1.

Список пар.

```
nums = (2, 5, 8, 10, 15)
print([(x + 2, x * 3) for x in nums])
# [(4, 6), (7, 15), (10, 24), (12, 30), (17, 45)]
```

Реализация 2.

Список значений.

```
nums = (2, 5, 8, 10, 15)
print([x + 2 for x in nums] + [x * 3 for x in nums])
# [4, 7, 10, 12, 17, 6, 15, 24, 30, 45]
```

Реализация 3.

Множество значений (только уникальные значения)

```
nums = (2, 5, 8, 10, 15)
print({v for x in nums for v in (x+2, x*3)})
# {4, 6, 7, 15, 10, 24, 12, 30, 17, 45}
```

Список литературы

[1] «PEP 8 -- Style Guide for Python Code | Python.org,» [В Интернете]. Available: https://www.python.org/dev/peps/pep-0008/. [Дата обращения: 05 09 2021].

[2] «Все, что нужно знать о сборщике мусора в Python / Habr,» [В Интернете]. Available: https://habr.com/ru/post/417215/. [Дата обращения: 02 09 2021].

[3] «Дополнительный код,» [В Интернете]. Available: https://ru.wikipedia.org/wiki/Дополнительный_код. [Дата обращения: 03 09 2021].

[4] «8. Compound statements — Python 3.9.7 documentation,» [В Интернете]. Available: https://docs.python.org/3/reference/compound_stmts.html#grammar-token-if-stmt. [Дата обращения: 05 09 2021].

[5] «8. Составные операторы,» [В Интернете]. Available: https://digitology.tech/docs/python_3/reference/compound_stmts.html#else. [Дата обращения: 05 09 2021].

[6] «PEP 20 -- The Zen of Python | Python.org,» [В Интернете]. Available: https://www.python.org/dev/peps/pep-0020/. [Дата обращения: 05 09 2021].

[7] «Download Python | Python.org,» [В Интернете]. Available: https://www.python.org/downloads/. [Дата обращения: 05 09 2021].

[8] «Программная ошибка - ВикипедиЯ,» [В Интернете]. Available: https://ru.wikipedia.org/wiki/Программная_ошибка. [Дата обращения: 05 09 2021].

[9] «Стек вызовов - Википедия,» [В Интернете]. Available: https://ru.wikipedia.org/wiki/Стек_вызовов. [Дата обращения: 06 09 2021].

[10] «Built-in Types — Python 3.9.7 documentation,» [В Интернете]. Available: https://docs.python.org/3/library/stdtypes.html#common-sequence-operations. [Дата обращения: 10 09 2021].

[11] «List of Unicode Characters of Category “Decimal Number”,» [В Интернете]. Available: https://www.compart.com/en/unicode/category/Nd. [Дата обращения: 11 09 2021].

[12] «List of Unicode Characters of Category “Other Number”,» [В Интернете]. Available: https://www.compart.com/en/unicode/category/No. [Дата обращения: 11 09 2021].

[13] «Юникод - Википедия,» [В Интернете]. Available: https://ru.wikipedia.org/wiki/Юникод. [Дата обращения: 11 09 2021].

[14] «2. Lexical analysis — Python 3.9.7 documentation,» [В Интернете]. Available: https://docs.python.org/3/reference/lexical_analysis.html#string-and-bytes-literals. [Дата обращения: 11 07 2021].

[15] «PEP 3107 -- Function Annotations | Python.org,» [В Интернете]. Available: https://www.python.org/dev/peps/pep-3107/. [Дата обращения: 21 09 2021].

[16] «Wayback Machine,» [В Интернете]. Available: http://web.archive.org/web/20070730120117/http://oakwinter.com/code/typecheck/. [Дата обращения: 21 09 2021].

[17] «PEP 257 -- Docstring Conventions | Python.org,» [В Интернете]. Available: https://www.python.org/dev/peps/pep-0257/. [Дата обращения: 21 09 2021].

[18] «reStructuredText,» [В Интернете]. Available: https://docutils.sourceforge.io/rst.html. [Дата обращения: 21 09 2021].

[19] «PEP 287 -- reStructuredText Docstring Format | Python.org,» [В Интернете]. Available: https://www.python.org/dev/peps/pep-0287/. [Дата обращения: 21 09 2021].

[20] «PEP 570 -- Python Positional-Only Parameters | Python.org,» [В Интернете]. Available: https://www.python.org/dev/peps/pep-0570/. [Дата обращения: 21 09 2021].

[21] «ЕГЭ по информатике: генератор вариантов,» [В Интернете]. Available: https://kpolyakov.spb.ru/school/ege/gen.php?action=viewAllEgeNo&egeId=24&cat155=on&cat156=on&cat164=on. [Дата обращения: 17 09 2021].

[22] «Итерируемый объект, итератор и генератор в Python,» [В Интернете]. Available: https://pythoner.name/iterator. [Дата обращения: 11 09 2021].