

# Справочник по языку Python с примерами

2012-2018

# Оглавление

<b>0</b>	<b>Об этом справочнике</b>	<b>3</b>
<b>1</b>	<b>Обзор основных свойств языка</b>	<b>4</b>
1.1	Какие бывают языки	4
1.2	Значения в языке Python	5
1.3	Идентификаторы	6
1.4	Иерархия типов в Питоне	6
1.5	Операции и выражения	6
1.6	Приведение числовых типов	8
<b>2</b>	<b>Основные операторы</b>	<b>9</b>
2.1	Оператор присваивания	9
2.2	Функция <code>print</code>	10
2.3	Форматированный вывод	10
2.4	Функция ввода: <code>input()</code>	13
2.5	Условный оператор	13
2.6	Оператор цикла <code>while</code>	15
2.7	Операторы <code>break</code> и <code>continue</code>	16
2.8	Комментарии	17
<b>3</b>	<b>Строки, оператор цикла <code>for</code></b>	<b>18</b>
3.1	Строковые значения	18
3.2	Операции со строками	18
3.3	Оператор цикла <code>for</code>	21
<b>4</b>	<b>Списки, функция <code>range</code>, оператор цикла <code>for</code> (продолжение)</b>	<b>22</b>
4.1	Значения списков	22
4.2	Операции со списками	22
4.3	Присваивание и копирование списков	24
4.4	Функция <code>range</code>	25
4.5	Оператор цикла <code>for</code>	25
4.6	Функция <code>random</code>	27
4.7	Операции с вложенными списками	27
<b>5</b>	<b>Кортежи</b>	<b>30</b>
5.1	Свойства кортежей	30
5.2	Значения кортежей	30
5.3	Операции с кортежами	31
5.4	Применение кортежей	31
<b>6</b>	<b>Словари</b>	<b>32</b>
6.1	Создание словаря	32
6.2	Получение значений словаря, изменение словаря	32
6.3	Слияние словарей	33
<b>7</b>	<b>Множества</b>	<b>35</b>
7.1	Создание множества и логические операции с ними	35
7.1.1	Операции над множествами, не приводящие к их изменению	36
7.1.2	Изменение множеств	36

<b>8</b>	<b>Встроенные функции и методы</b>	<b>38</b>
8.1	Числовые функции . . . . .	38
8.2	Обработка последовательностей . . . . .	39
8.2.1	Общие функции . . . . .	39
8.2.2	Методы работы со списками . . . . .	39
8.2.3	Методы работы со строками . . . . .	42
<b>9</b>	<b>Работа с файлами</b>	<b>45</b>
9.1	Режимы работы с файлами. . . . .	45
9.2	Чтение из файла . . . . .	45
9.3	Запись в файл . . . . .	47
<b>10</b>	<b>Динамическая типизация</b>	<b>48</b>
10.1	Переменные, объекты и ссылки . . . . .	48
10.2	Автоматическая сборка мусора . . . . .	48
10.3	Разделяемые ссылки, изменяемые и неизменяемые типы . . . . .	49
<b>11</b>	<b>Функции, параметры, вызовы функций.</b>	<b>51</b>
11.1	Определение функции . . . . .	51
11.2	Вызов функции, значение None . . . . .	52
11.3	Запуск программы, содержащей функции . . . . .	54
11.4	Блоки, области видимости имён, локальные и глобальные переменные . . . . .	55
11.5	Передача переменных изменяемых и неизменяемых типов в качестве параметров функций . . . . .	57
11.6	Рекурсивные функции . . . . .	59

# Глава 0

## Зачем это было написано

Этот текст задумывался как краткий справочник по языку Python, снабжённый большим количеством примеров. Несмотря на явное указание того, что интерпретатор Python выдаёт в качестве ответа, имеет смысл выполнять указанные фрагменты кода в интерпретаторе. Во-первых, вы попытаетесь проверить собственные гипотезы, которые не нашли отражения в тексте, во-вторых — отыщите опечатки.

Этот справочник не предназначен для знакомства с программированием "с нуля". Скорее может помочь записать на языке Python уже понятные конструкции и познакомить с основными отличиями от Паскаля или Си.

Обо всех опечатках, ошибках, неудачных примерах и нелогичной последовательности изложения сообщайте по адресу: [gusarer@gmail.com](mailto:gusarer@gmail.com)

# Глава 1

## Обзор основных свойств языка. Значения, типы, выражения. Приведение ТИПОВ.

### 1.1 Какие бывают языки

- Высокоуровневые и низкоуровневые  
Текст программы на языке Питон не похож на машинный код, программа не оперирует непосредственно содержимым регистров процессора. Его служебные слова — это слова английского языка; язык, с помощью которого записываются выражения почти полностью совпадает с принятым в математике алгебраическим языком.
- Общего назначения и специализированные  
На Питоне можно написать графический редактор, вебсайт, программу, обрабатывающую результаты опытов, игру в крестики-нолики и многое другое.
- Интерпретируемые и компилируемые  
Программу на Питоне выполняет программа-интерпретатор. Почти для каждой операционной системы и архитектуры процессора существует свой интерпретатор, так что текст программы не требуется модифицировать. Правда, из-за необходимости обработки интерпретатором программа на Питоне работает медленнее, чем аналогичная программа на компилируемом языке (C++, Паскаль).

Определение языка включает в себя синтаксические и семантические правила. Первые описывают структуру, вторые — смысл.

- Синтаксис — определяет разрешённые конструкции языка  
`1x = 6 # имя переменной не может начинаться с цифры`  
`Аткрой акно! # нет таких слов в русском языке`

Синтаксические ошибки всегда обнаруживаются до выполнения программы в компилируемых языках. В интерпретируемых, как правило, на этапе выполнения, но, вообще говоря, бывают и предварительные проверки до выполнения программы интерпретатором. Эти ошибки — самые простые.

- Статическая семантика — связывает синтаксически верные конструкции общим смыслом

```
x = 6 + 'asd' # операция не определена для int и string
print(t) # имя t не определено (не имеет значения)
```

Такие ошибки обнаруживаются на этапе выполнения программы. Пример программы, которая при разных входных данных либо корректно завершается, либо выдаёт сообщение об ошибке:

↓

```
x = int(input())
if x > 1:
    print(5)
else:
    print('asd' // x) # не определена операция деления строки на число
```

- Самые сложные ошибки возникают, когда программа написана верно в предыдущих двух смыслах, но на некоторых входных данных не заканчивает работу или выдаёт неправильный ответ.

↓

```
def power(a, n):
    res = 1
    i = 1
    while i < n:
        res = res * a
    return res
```

В этом примере цикл `while` никогда не закончит работу, если переданное значение параметра `n` больше 1.

Для естественного языка примером может послужить сочинение на заданную тему, заслужившее пометку «Тема не раскрыта».

*Условия реализации основной образовательной программы среднего (полного) общего образования должны обеспечивать для участников образовательного процесса возможность: . . . формирования у обучающихся . . . , готовности к защите Отечества, службе в Вооружённых Силах Российской Федерации.*

(Выдержка из Федерального Государственного Образовательного Стандарта Общего Образования)

## 1.2 Значения в языке Python

- Числа
  - Для хранения целых чисел Python в отличие от большинства языков (C++, Паскаль) использует всю доступную память. Фактически, работа с длинными целыми числами реализована в стандарте языка.
  - Вещественные числа реализованы на основе чисел с плавающей точкой двойной точности — `double` (64 бита). 1 бит на знак, 11 бит на показатель экспоненты и 52 бита на значащую часть (мантиссу).  
Примеры: `3.0`, `-123.345`, `.76543`, `23.490e23`.
- Логические: `True`, `False`. Логический тип на самом деле является лишь подтипом целого, значение `False` соответствует нулю, `True` — любому ненулевому целому числу.
- Упорядоченные последовательности
  - **строки**: последовательность литералов (символов). Строковые значения должны быть заключены в одинарные и двойные кавычки. Примеры: `'a'`, `'abc'`, `'234g 3654__'`, `"don't"`.
  - **списки**: последовательность произвольных элементов, разделяемых запятыми и взятая в квадратные скобки. Пустой список — `[]`. Примеры: `[1, 2, 3]`, `['Name', 'Surname', 5]`.
  - **кортежи**: последовательность произвольных элементов, разделяемых запятыми, которая может быть взята в круглые скобки. Пустой кортеж обязательно должен быть взят в скобки: `()`, кортеж из одного элемента обязательно должен содержать запятую после единственного элемента: `(4,)`.  
Примеры: `(2, 3)`, `('abc', 345)`

## 1.3 Идентификаторы

Идентификатором в Питоне называется последовательность заглавных латинских букв, строчных латинских букв, знаков подчёркивания и цифр, начинающаяся не с цифры и отличная от служебного слова.

### Служебные слова

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

Идентификаторы чувствительны к регистру: `count` и `Count` — разные идентификаторы.

Примеры идентификаторов: `max`, `next`, `a1`, `_2`, `count`, `Count`, `search_appropriate_element`

## 1.4 Иерархия типов в Питоне

### 1. None

### 2. Числа

- Целые
- Вещественные

### 3. Последовательности

- Строки
- Списки
- Кортежи

### 4. Отображения

- Словари

## 1.5 Операции и выражения

### • Операции с целыми числами

- унарные арифметические: изменение знака (-)
- бинарные арифметические: сложение (+), вычитание (-), умножение (\*), целочисленное деление (//), остаток от целочисленного деления (%), возведение в степень (\*\*)
- унарные битовые: инверсия (~)
- бинарные битовые: сдвиг вправо (>>), сдвиг влево (<<), битовое «И» (&), битовое «ИСКЛЮЧАЮЩЕЕ ИЛИ» (^), битовое «ИЛИ» (|)

### • Операции с вещественными числами

те же, что и арифметические операции для целых чисел, но вещественное деление обозначается иначе (/), а остаток от деления (%) расширяется для вещественных чисел естественным образом:

↓

```
print(3.14159 % 2) # результат вычисления равен 1.14159, т.к. 1.14159, С,.Рс. 3.1159 = 1 * 2 + 1.14159
print(3.14159 % 1.2) # результат вычисления равен 0.74159, т.к. 0.74159, С,.Рс. 3.14159 = 2 * 1.2 + 0.74159
```

### • Логические операции

- бинарная операция логическое «И» (and): результат выполнения равен `True` тогда и только тогда, когда оба операнда равны `True`, в противном случае результат выполнения равен `False`.
- бинарная операция логическое «ИЛИ» (or): результат выполнения равен `True` тогда и только тогда, когда хотя бы один операнд равен `True`, в противном случае результат выполнения равен `False`.
- унарная операция логическое «НЕ» (not): результат выполнения равен `True`, если операнд равен `False`, в противном случае результат выполнения равен `True`.

**Числовые выражения** в Питоне конструируются при помощи числовых значений, идентификаторов, знаков операций и круглых скобок в принятом в алгебре порядке.

Примеры:

↓

```
print(123 // 3 // 4 + 5 % 7) # значение этого выражения - целое число (15),
                             # т.к. все операнды - целые,
                             # и результат всех промежуточных операций - тоже целое число
print((a + b) * (a - b))    # если хотя бы одно значение (a или b) - вещественное число,
                             # то результат - вещественное число, иначе - целое число
print(-c // (a + b) * (a - b)) # то же, что и в предыдущем случае

# в этом примере результат будет в любом случае вещественным числом,
# т.к. последней выполняется операция вещественного деления ( / )
print((dist1 + dist2 + dist3) / (time1 + time2 + time3))
```

К значениям числовых типов можно применять операции сравнения, результатом выполнения которой является значение логического типа:

Операция	Описание
<	строго меньше
<=	меньше или равно
>	строго больше
>=	больше или равно
==	равно
!=	не равно

Примеры использования операций сравнения и логических операторов:

↓

```
b * b - 4 * a * c >= 0
(x < y) and (x - 4) * (y - 3) <= 25 and y <= 6 - x * x
```

Сравнения можно объединять в цепочки. Например, запись `a<b<c` разрешена синтаксисом языка Python и имеет общепринятый смысл. Такая запись эквивалентна следующей: `a<b and b<c` с той разницей, что `b` вычисляется **один** раз. Сравнения выполняются слева направо и если не выполнено хотя бы одно, то результат сравнения равен `False`.

### Приоритет выполнения операций в выражениях (по убыванию приоритета):

Операция	Описание
()	скобки
**	возведение в степень
+x, -x, ~x	унарный плюс, унарный минус, битовое НЕ
*, /, %	умножение, деление, остаток от деления
+, -	сложение, вычитание
<<, >>	битовые сдвиги
&	битовое И
^	битовое исключающее ИЛИ
	битовое ИЛИ
in, not in, is, is not, <, <=, >, >=, <>, !=, ==	проверка на принадлежность, на идентичность, сравнения
not	логическое НЕ
and	логическое И
or	логическое ИЛИ

Соответствующий раздел документации: <https://docs.python.org/3/reference/expressions.html>.

Все операции, имеющие одинаковый приоритет выполняются слева направо. Исключениями являются сравнения (см. цепочки сравнений выше) и операция возведения в степень, которая выполняется справа налево. Таким образом `2**3**2` понимается как `2**(3**2)`.

#### Замечания

- Проверка на равенство для вещественного типа не имеет смысла.

Дело в том, что в отличие от целых чисел, для которых существует (и единственное) разложение по степеням двойки, не для всех десятичных дробей существует представление в виде конечной суммы дробей вида  $\frac{1}{2^n}$ .

Например,  $\frac{1}{7} = 0.00(100)$

Вынужденное отбрасывание старших разрядов (округление) приводит, например, к таким результатам (сравнение  $\sqrt{2} \cdot \sqrt{2}$  и 2):



↓

```
import math
a = math.sqrt(2)
print(a * a == 2)      # результат выполнения False
```

## 1.6 Приведение числовых типов

Как Python определяет тип числового выражения, если в таком выражении могут встретиться значения и переменные разных типов? Это может быть сделано двумя разными способами — или автоматически (такой способ называется неявным приведением типа), или путём явного указания типа, который должен иметь результат вычисления выражения.

- **Неявное приведение**

Тип числового выражения определяется типом наиболее «сложного» из входящих значений и переменных. Числовые типы упорядочены по сложности так: целый < вещественный.

Примеры:

- выражения целого типа: целые операнды и операции, результат которых — целое число  
`1 // 3, 2 * 30, 5 // (5 % 3)`
- выражения вещественного типа: содержат по крайней мере один вещественный операнд или операцию, результат которой — вещественное число:  
`1 / 5, 45 % 23 + 0.5, 3 + 6 - 6 / 3`

- **Явное приведение**

Для явного приведения выражения к нужному типу необходимо использовать встроенные функции `int()`, `float()`. Каждая зависит от одного аргумента и переводит его в целый и вещественный тип соответственно.

Примеры корректных вызовов: `int(2 / 2)`, `float(7 // 4)`

Примеры некорректных вызовов: `float(-2 ** 2000)`

## Глава 2

# Основные операторы: присваивания, ввод/вывод, условный, оператор цикла while, операторы break и continue

## 2.1 Оператор присваивания

Синтаксис оператора присваивания:

```
<переменная> <команда присваивания> <выражение>
```

где <переменная> — идентификатор

<команда присваивания> — одна из команд =, +=, -=, \*=, /=, %=

<выражение> — выражение любого типа.

Команды +=, -=, \*=, /=, %= являются сокращённой формой записи следующих команд:

↓

```
x = 7
x += 2 # увеличение значения переменной x на 2: x = x + 2
x -= 2 # уменьшение значения переменной x на 2: x = x - 2
x *= 2 # увеличение значения переменной x в 2 раза: x = x * 2
x /= 2 # уменьшение значения переменной x в 2 раза: x = x // 2
x / 2 # вещественное деление
x %= 2 # получение остатка от деления x на 2: x = x % 2
```

Оператор присваивания выполняет два действия: сначала вычисляет значение выражения, стоящего в правой части, затем связывает имя переменной, стоящей в левой части с вычисленным значением. Тип переменной, стоящей в левой части определяется типом вычисленного значения.

До того, как имя переменной было связано с каким-то значением, её нельзя использовать в правой части оператора присваивания:

↓

```
a = b + 3 # невозможно вычислить значение выражения b + 3, т.к. неизвестно чему равно b
t += 7    # так нельзя, это то же самое, что и t = t + 7, невозможно вычислить t + 7
a = 13
a *= 3    # переменная a теперь имеет значение 39
```

↓

```
x = 3      # значение переменной x равно 3 (целый тип)
x = 5 + x  # значение переменной x равно значению выражения 5 + x
           # переменная x теперь имеет значение 8 (целый тип)
x = x * 2.2 # значение переменной x равно значению выражения x * 2.2
           # переменная x теперь имеет значение 17.6 (вещественный тип)

b = 2
b = b / 3  # значение переменной b равно 0.66 (вещественный тип)

x = 2.6
# сейчас переменная x имеет вещественный тип, её значение равно 2.6
x = (x > 3) and (x * x < 100)
# значение выражения в правой части оператора присваивания,
# а значит и переменная x имеют логический тип и равны False
```

Оператор присваивания имеет несколько расширений. Одно из них следующее: левая часть оператора может содержать несколько имён переменных, разделённых запятой. Правая часть оператора в таком случае должна содержать столько же выражений, также разделённых запятой. Выражения в правой части вычисляются, затем получившиеся значения присваиваются перечисленным в левой части переменным, слева направо.

↓

```
# a присваивается значение целого типа (4)
# b присваивается значение строкового типа ('abc')
a, b = 4, 'abc'
# а так можно записать обмен значениями для двух переменных
a, b = b, a
```

В подобном операторе присваивания важно понимать последовательность вычисления выражений в правой части и присваиваний: присваивание3, присваивание4 = вычисление1, вычисление2

## 2.2 Функция print

Синтаксис: `print(<выражение>[, <выражение>])`

Оператор `print` выводит на экран вычисленные значения указанных выражений, затем символ окончания строки. Если надо вывести значения нескольких выражений в одной строке, их следует перечислить через запятую.

↓

```
print(34 / 4)           # результат 8.5
print(34 % 4)          # результат 2
print(34 // 4, 34 % 4) # результат 8 2
```

По умолчанию функция `print` переводит курсор в начало следующей строки. Это происходит потому что у функции `print` есть необязательный параметр `end`, равный по умолчанию символу переноса строки. Функции `print` передать значение этого параметра явно:

↓

```
print('quotient:', 34 / 4, end='')
print(', remainder:', 34 % 4)
```

```
# результат выполнения:
quotient:8, remainder:2
```

При выводе нескольких значений функция `print` разделяет их одним пробелом. Управляет этим необязательный параметр `sep`. По умолчанию его значение равно строке из одного пробела. Например:

↓

```
a = 4
print('Square of', a, 'is', a * a, '.') # результат: Square of 4 is 16.
```

Чтобы избавиться от ненужного пробела перед точкой, нужно воспользоваться параметром `sep` функции `print`:

↓

```
a = 4
print('Square of ', a, ' is ', a * a, '.', sep = ' ') # результат: Square of 4 is 16.
```

## 2.3 Форматированный вывод

Если программа нетребовательна к формату выводимых значений (например нужно вывести несколько чисел или иных значений, разделённых пробелом), разумно пользоваться стандартной функцией `print`, перечисляя значения через запятую. Но часто результат работы программы требуется выводить в строго определённом формате, причём параметров `sep` и `end` может быть недостаточно. Например при выводе вещественных чисел необходимо точно указать количество символов, отводимых под дробную часть. Для этих (и для многих других) целей служит т.н. форматированный вывод.

Для задания выводимым на экран данным требуемого формата можно использовать функцию `print` в сочетании с методом `str.format()`:

```
print(<строка форматирования>.format(<перечень форматируемых выражений>))
```

Строка форматирования содержит текст и спецификации формата, взятые в фигурные скобки `{ }`. Спецификация формата может быть пустой, содержать номер выражения из списка форматируемых выражений (которые нумеруются, начиная с нуля) или номер выражения со спецификацией формата. Весь текст, находящийся вне этих скобок, выводится без изменений. Приведём несколько примеров с результатами вывода:

↓

```

a = 3
b = 4
print('{0}{1}{2}'.format(a, b, a * b))
3412 # между спецификациями формата отсутствуют пробелы
print('{0} {1} {2}'.format(a, b, a * b))
3 4 12
print('{}-{}-{}'.format(a, b, b / a))
# ошибка: количество спецификаций формата превышает количество форматируемых выражений
print('a={0}, b={1} a % b = {3}, a // b = {2}'.format(a, b, a // b, a % b))
a = 3, b = 4, a % b = 3, a // b = 0
# можно явно указать соответствие спецификаций выражениям

```

Несколько слов о спецификациях формата для вывода чисел.

Спецификация формата для вывода целых чисел выглядит так:

`<k>d`

где `<k>` это целое число, обозначающее минимальное количество символов, которое должен занимать вывод значения. Если параметр `<k>` не указан, то вывод будет занимать столько символов, сколько фактически требуется.

Спецификация формата для вывода вещественных чисел:

`<all>.<fraction>f`

где `<fraction>` — количество символов, отводимое на вывод дробной части числа, `<all>` — суммарное количество символов, отводимое на вывод всего числа, включая десятичную точку и знак.

- `%f` — если не указан ни один из параметров, то по умолчанию на вывод дробной части отводится 6 знаков, а на вывод всего значения — сколько потребуется
- `%10f` — если указан только один параметр (нет точки и второго параметра), то этот параметр интерпретируется как `<all>`, а на вывод дробной части отводится (по умолчанию) 6 знаков
- `%.5f` — если не указан `<all>`, то на вывод дробной части отводится указанное количество символов, а на всё значение — сколько потребуется
- `%5.f` — если не указан `<fraction>`, то на вывод всего значения отводится указанное количество символов, а параметр `<fraction>` полагается равным нулю, т.е. дробная часть не выводится вовсе

Ниже приведены примеры использования форматированного вывода с комментариями:

↓

```
print('{0:d} {1:d}'.format(12, 144))
# на каждое значение отводится столько символов, сколько требуется для вывода
# выводимые числа разделяет один пробел, указанный между спецификациями формата
12 144

print('{0:5d}{1:5d}{2:5d}'.format(2, 2 ** 10, 2 ** 20))
# на каждое значение отводится 5 символов, если их хватает
# если не хватает - то столько, сколько надо
# пробелов между выводимыми значениями нет, потому второе и третье "слиплись"
 2 10241048576

print('{0:8d}{1:8d}{2:8d}'.format(2, 2 ** 10, 2 ** 20))
# если отвести на каждое значение заведомо достаточное количество символов (здесь 8)
# то результат станет лучше:
 2   1024 1048576

x = 11 / 7
print('{0:5.3f} {1:5.3f} {2:10.5f}'.format(x, x * x, x * x * x))
# на первые два значения отводится 5 знаков на всё значение и 3 знака на дробную часть
# на последнее значение отводится 10 знаков на всё и 5 знаков на дробную часть
1.571 2.469  3.88047

import math
a = math.sqrt(2)
b = math.sqrt(3)
print('a = {0:5.6f}, b = {1:5.6f}'.format(a, b))
# обратите внимание на пробелы, окружающие знаки равенства
a = 1.414214, b = 1.732051

# таблица умножения
for i in range(1, 11):
    for j in range(1, 11):
        print('{0:4d}'.format(i * j), end = ' ')
    print()

# результат выполнения:
 1  2  3  4  5  6  7  8  9 10
 2  4  6  8 10 12 14 16 18 20
 3  6  9 12 15 18 21 24 27 30
 4  8 12 16 20 24 28 32 36 40
 5 10 15 20 25 30 35 40 45 50
 6 12 18 24 30 36 42 48 54 60
 7 14 21 28 35 42 49 56 63 70
 8 16 24 32 40 48 56 64 72 80
 9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

Подробнее про форматированный вывод и все его параметры можно прочитать в соответствующем разделе документации:

<http://docs.python.org/3/library/string.html#format-specification-mini-language>

## 2.4 Функция ввода: input()

Синтаксис: `<имя> = input(<s>)`

Функция `input` выводит на экран строку `<s>`, затем считывает введённую пользователем строку и связывает `<имя>` с этим строковым значением. Примеры использования:

↓

```
d = 4
s = input('number:')
print(d * s)
print(d * int(s))
print(d / float(s))

# вывод программы, если пользователь ввёл строку '2':
'2222'
8
2.0

# вывод программы, если пользователь ввёл строку '3.14'
'3.143.143.143.14'
ValueError: invalid literal for int() with base 10: '3.14'
1.2738853503184713

# вывод программы, если пользователь ввёл строку 'abc':
'abcabcabcabc'
ValueError: invalid literal for int() with base 10: 'abc'
ValueError: could not convert string to float: 'abc'
```

Если известно что на вход программе будет подано число, можно организовать его ввод следующим образом:

↓

```
x = int(input()) # если на вход подаётся целое число
y = float(input()) # если на вход подаётся вещественное число
```

## 2.5 Условный оператор

Сокращённый синтаксис	Расширенный синтаксис	Полный синтаксис
<pre>if &lt;условие&gt;:     &lt;составной оператор&gt;</pre>	<pre>if &lt;условие&gt;:     &lt;составной оператор&gt; else:     &lt;составной оператор&gt;</pre>	<pre>if &lt;условие&gt;:     &lt;составной оператор&gt; elif &lt;условие&gt;:     &lt;составной оператор&gt; ... elif &lt;условие&gt;:     &lt;составной оператор&gt; else:     &lt;составной оператор&gt;</pre>

где `<условие>` — выражение, имеющее значение логического типа или неявным образом приводимое к нему  
`<составной оператор>` — один или несколько операторов языка Питон, называемые телом условного оператора

Как интерпретатор Python понимает, где заканчивается тело условного оператора? Все строчки, содержащие операторы тела начинаются с отступа, который обычно составляет 4 пробела. При этом отступ считается относительно стартовой позиции заголовка условного оператора (содержащего служебное слово `if`). Это же правило распространяется на вложенные конструкции:

```
if <условие>:
...<оператор>
...
...<оператор>
...if <условие>:
.....<оператор>
...
.....<оператор>
...<оператор> # продолжается выполнение первого условного оператора
...
...<оператор>
```

Проверка условия делается один раз, перед выполнением тела условного оператора. Так, в примере:

↓

```
n = 2
if n < 4:
    n = n * 2
    n = n * 2
print(n)
```

программа выведет 8 (т.е. выполнятся все операторы, составляющие тело оператора `if`).

Пример расширенного синтаксиса условного оператора:

↓

```
# вывод трёх введённых значений в убывающем порядке
# для вложенных условных операторов отступы отсчитываются от его заголовка
x = input()
y = input()
z = input()
if x > y:
    if y > z:
        print(x, y, z)
    else:
        if x > z:
            print(x, z, y)
        else:
            print(z, x, y)
else:
    if x > z:
        print(y, x, z)
    else:
        if y > z:
            print(y, z, x)
        else:
            print(z, y, x)
```

Условный оператор позволяет избежать выполнения частей программы в зависимости от истинности какого-то логического выражения. Таким образом, время выполнения программы, составленной из известных нам операторов ввода-вывода и условного остаётся постоянным вне зависимости от объёма входных данных. Каждый оператор выполнится не более одного раза.

## 2.6 Оператор цикла while

Оператор цикла `while` имеет два варианта синтаксиса:

Стандартный синтаксис	Расширенный синтаксис
<pre>while &lt;условие&gt;:     &lt;составной оператор&gt;</pre>	<pre>while &lt;условие&gt;:     &lt;составной оператор&gt; else:     &lt;составной оператор&gt;</pre>

где `<условие>` — выражение, имеющее значение логического типа

`<составной оператор>` — один или несколько операторов, которые называются телом цикла.

Однократное выполнение тела цикла называется итерацией цикла. Операторы, составляющие тело цикла, записываются с отступом относительно заголовка цикла, аналогично тому, как записывается условный оператор.

Тело цикла `while` выполняется до тех пор, пока значение логического выражения `<условие>` равно `True`. Таким образом, цикл может выполниться *любое* количество раз, от нуля (если значение логического выражения равно `False` перед началом выполнения цикла) до бесконечности (когда значение логического выражения всегда `True`).

В расширенном синтаксисе оператора `while`: если значение `<условия>` стало равным `False` выполняется составной оператор после служебного слова `else`.

Важно понимать, что проверка условия делается всегда **перед** началом (очередного) выполнения тела цикла. В процессе выполнения операторов тела цикла условие не контролируется.

Как правило, цикл `while` используется, когда заранее неизвестно, сколько раз должно выполниться тело цикла, зато известно условие, при выполнении которого должен выполняться этот цикл.

↓

```
# до окончания выполнения цикла неизвестно, сколько итераций будет выполнено
# здесь мы ждём, пока пользователь не введёт строку 'yes' или строку 'no'
s = input('Enter yes/no:')
while s != 'yes' or s != 'no':
    s = input('Enter yes/no:')
print('thank you, you have entered: ' + s)

# наибольшее целое число, чей квадрат не превосходит x
x = int(input())
res = 0
while (res + 1) * (res + 1) <= x:
    res += 1
print(res)

# ввод последовательности целых чисел, заканчивающейся нулём
# нахождение порядкового номера максимального элемента
k = int(input())
cnt = 1
max = k
max_n = cnt
while k != 0:
    if k > max:
        max = k
        max_n = cnt
    cnt += 1
    k = int(input())
print(max_n)
```



↓

```
# ввод последовательности целых чисел, заканчивающейся нулём
# нахождение среднего арифметического последовательности
k = int(input())
cnt = 0
s = 0
while k != 0:
    s = s + k
    cnt = cnt + 1
    k = int(input())
print(s / cnt)

# определение максимальной цифры в десятичной записи натурального числа
# решение не предполагает предварительного вычисления количества разрядов
# или перевода числа в строковый тип
n = int(input())
mx = n % 10
n //= 10
while n > 0:
    if n % 10 > mx:
        mx = n % 10
    n //= 10
print(mx)
```

Казалось бы, какой смысл в расширенном синтаксисе оператора `while`? Поместим операторы из части `else` сразу после цикла, и добъёмся ровно такого же поведения программы. Разница станет ясна после знакомства с оператором `break` в параграфе 2.7.

## 2.7 Операторы `break` и `continue`

В Python существует два дополнительных способа управлять работой цикла — прерывать выполнение оператора цикла или прерывать выполнение текущей итерации.

Первый — это оператор `break`. Этот оператор прерывает выполнение текущей итерации и выходит цикла, после чего программа выполняет следующий после цикла оператор. Условие цикла после выполнения оператора `break` не проверяется. Кроме того, не выполняется и составной оператор, соответствующий `else` (если он есть). Таким образом, оператор `break` прекращает работу **всего** оператора цикла.

Например:

↓

```
# как только сумма введённых пользователем чисел равна нулю
# вывести их количество и значение максимальной (положительной) суммы
k = int(input())
s = 0
maxs = 0
count = 0
while True:
    s += k
    if s > maxs:
        maxs = s
    count += 1
    if s == 0:
        break
    k = int(input())
print(count, maxs)
```

А вот пример использования расширенного синтаксиса оператора `while`:

↓

```
# программа предлагает пользователю угадать число от 1 до 20
# для прекращения работы с программой пользователь может ввести отрицательное число
from random import randint
n = 20
to_be_guessed = randint(1, 20)
guess = 0
while guess != to_be_guessed:
    guess = int(input("New number: "))
    if guess > 0:
        if guess > to_be_guessed:
            print("Number too large")
        else:
            print("Number too small")
    else:
        print("Sorry, that you're giving up!")
        break
else:
    print('Congratulation!')
```

Второй способ — оператор `continue`. Его отличие от оператора `break` в том, что, заканчивая выполнение текущей итерации, он не завершает выполнение самого цикла, а переходит к проверке его условия. Затем, в зависимости от результата проверки, переходит к выполнению следующей итерации.

↓

```
x = -50
while x < 50:
    if x == 0:
        continue
    print(1 / x)
```

## 2.8 Комментарии

Комментариями в Python называются строки или части строк программы, начинающиеся с символа `'#'`. Такие строки не воспринимаются интерпретатором Python, как команды, а игнорируются. Комментарии используются для описания на естественном языке нетривиальных (неочевидных) частей программы.

Ещё один способ повысить читаемость программы — давать переменным имена, отражающие смысл (а иногда и тип) этих переменных. Не всегда разумно давать *всем* переменным длинные, «говорящие» названия. Но в любом случае стоит комбинировать оба эти метода (комментарии и имена переменных), чтобы программа оставалась понятной вне зависимости от времени, прошедшего с момента её написания.

Пример:

↓

```
k = 2
n = int(input())
while n > 2:
    while (n % k) == 0: # пока можем делить на k
        print(k, end=' ') # выписываем
        n = n // k        # и делим
    k = k + 1
```

# Глава 3

## Строки, оператор цикла for

Строками в Python называются упорядоченные последовательности символов.

### 3.1 Строковые значения

Строковое значение записывается как последовательность символов, заключённая в одинарные или двойные кавычки:  
↓

```
s = 'abcdefgh'
f = s[2]      # значение переменной f равно 'c'
print(s[0])  # будет выведен символ 'a'
```

Далее в тексте все строковые значения будут братья в кавычки. Фраза *переменная a имеет значение 'qwerty'* будет означать, что строка **a** состоит из символов, стоящих между кавычками. Для отдельных символов в Python нет специального типа, как в других языках программирования. Символ в Python — строка длины 1.

### 3.2 Операции со строками

- Ввод строки с клавиатуры Ввод строк описан выше, в главе 2.4.
- Операция индексирования строки:

<имя строки>[<выражение целого типа>]

Выражение в квадратных скобках называется индексом. Символы, составляющие строку нумеруются, начиная с нуля.

Например:

↓

```
s = 'abcdefgh'
f = s[2]      # значение переменной f равно 'c'
print(s[0])  # будет выведен символ 'a'
```

Значение отдельных символов строки менять нельзя, можно только создать новое значение и связать его с прежним именем. Эта особенность типа 'строка' называется неизменяемостью (immutable). Примеры:

↓

```
s = 'abcdefgh'
s[0] = 'z'    # это ошибка
s = 'zbcdefgh' # так можно
```

Значение индекса может быть отрицательным. Отрицательные индексы отсчитываются с конца строки в обратную сторону (от конца строки к её началу):

↓

```
s = 'abcdefgh'
print(s[-2]) # программа выведет символ 'g'
```

- **Функции ord и chr**

В памяти компьютера символы (и строки из них составленные) хранятся как числа (строки — как последовательности чисел). Взаимоднозначное соответствие между символами и числами даётся таблицей, которая называется ASCII (*American Standard Code for Information Interchange*).

В таблице 256 пар, исторически же используется только первая половина, содержащая основные символы: буквы латинского алфавита (строчные и прописные), цифры, знаки препинания и некоторые другие. Кроме перечисленных существуют т.н. управляющие символы с номерами, меньшими 32, которые не отображаются на экране.

Функция, которая по номеру в таблице возвращает символ, называется `chr()`, обратная ей — `ord()`. Вот некоторые примеры их использования:

↓

```
# программа выводит на экран символы с кодами от 32 до 126
k = 32
while k < 127:
    print(chr(k), end=' ')
    k += 1
```

- **Сравнение символов и строк**

Сравнение строк, состоящих из одного символа сводится к сравнению целых чисел — кодов этих символов, возвращаемых функцией `ord()`. Сравнение строк, состоящих из большего количества символов определяется лексикографическим порядком (как в словаре). Формально, лексикографический порядок для строк одинаковой длины `a` и `b` определяется следующим образом:

$$a < b \Leftrightarrow \exists k < \text{len}(a) : \forall i < k \text{ ord}(a[i]) = \text{ord}(b[i]), \text{ord}(a[k]) < \text{ord}(b[k])$$

Для определения лексикографического порядка строк разной длины надо в конец короткой строки мысленно дописать символы с кодом, меньшим кода любого возможного символа, затем применить стандартное правило.

↓

```
print('a' > 'c') # программа выведет False
print('1' < 'z') # программа выведет True
print('ab' < 'a') # программа выведет False
print('cba' > 'abc') # программа выведет True
print('123' < ' ') # программа выведет False
print('name' > '') # программа выведет True (пустая строка меньше любой непустой)

# функция возвращает True, если переданная строка состоит из одного символа-цифры
# и False в противном случае
def isdigit(s):
    if len(s) > 1:
        return False
    else:
        return '0' <= s <= '9'
```

Также для строк определены сравнения на равенство и неравенство.

- **Конкатенация (сложение) строк, функция len**

Функция `len(<строка>)` возвращает неотрицательное целое значение — количество символов в строке. Таким образом, операция индексирования, т.е. получения элемента строки `s` по его номеру определена для следующего диапазона значений:  $[-\text{len}(s), \text{len}(s) - 1]$ . При попытке получить значение символа по индексу, находящемуся вне этого диапазона произойдёт ошибка и программа прекратит работу.

↓

```
s = 'abcdefgh'
print(s[3]) # 'd'
print(s[-3]) # 'f'
print(s[len(s)]) # ошибка
print(s[-len(s)]) # 'a'
```

Операция сложения, определённая для числовых типов, может использоваться и со значениями строкового типа. Результатом сложения является новая строка, длина которой равна сумме длин слагаемых строк и составленная из их символов с сохранением порядка следования. Такая строковая операция называется *конкатенацией*.

↓

```
name = 'Petya'
sname = 'Petrov'
fname = name + ' ' + sname # значение переменной fname равно 'Petya Petrov'
print(len(name), len(sname), len(fname)) # будут напечатаны числа 5, 6 и 12
```

### • Умножение строки на целое число

Операция умножения, определённая для числовых типов может использоваться и со значениями строкового типа следующим образом:

$$\langle \text{переменная1} \rangle = \langle \text{переменная2} \rangle * \langle \text{переменная3} \rangle$$

где  $\langle \text{переменная1} \rangle$  и одна из переменных в правой части оператора присваивания — переменные строкового типа, а вторая переменная в правой части оператора присваивания — целого типа.

↓

```
x = 'a+b+'
s = x * 2 # значение переменной s равно 'a+b+a+b+'
s = 3 * x # значение переменной s равно 'a+b+a+b+a+b+'
```

### • Явное приведение к строковому типу, функция str()

Функция `str()` определена для аргумента любого типа. Сейчас эта функция интересна в связи с переводом в строковый тип числовых значений (целых и вещественных) и значений логического типа. Следующие примеры дают представление о принципе работы функции:

↓

```
# в каждой строке в комментарии указано значение переменной s после присваивания
k = 57.57
s = str(234) # '234'
s = str(k) # '57.57'
s = str(1.0/9) # '0.1111111111111111'
s = str(False) # 'False'
s = '(3 * 4 < ' + str(8) + ') == '+str(3*4<8) # '(3 * 4 < 8) == False'
# функцию str() без аргументов можно использовать для создания пустой строки
d = str() # значение d равно ''
```

Функция `str()` для каждого типа действует по некоторому стандартному правилу. Для перевода произвольного значения (например, числового) в строку в соответствии с заданным предписанием следует пользоваться форматированным выводом (см. раздел 2.3).

### • Операция in

Для строк определена операция `in` со следующим синтаксисом:

$$\langle s1 \rangle \text{ in } \langle s2 \rangle$$

где  $s1$  и  $s2$  - имена значений строкового типа. Выражение  $\langle s1 \rangle \text{ in } \langle s2 \rangle$  имеет логический тип и равно `True` если строка  $\langle s1 \rangle$  целиком входит в строку  $s2$ , начиная с какого-то места и `False` в противном случае. Например:

↓

```
p = 'abcdef'
f = 'bcd'
print(f in p) # программа выведет True
```

### • Сечения строк

В Python имеется удобное средство получения частей упорядоченных последовательностей, в частности строк: *срезы (или сечения)*. Синтаксис операции:

$$s[n: m: k]$$

где  $s$  — имя переменной строкового типа,  $n$ ,  $m$ ,  $k$  — выражения целого типа ( $k \neq 0$ ).

Значение указанного выражения составляется из символов с индексами равными  $x = n + i * k$ , где  $0 \leq i < \frac{m - n}{k}$ , причём в том же порядке, что и в строке  $s$ .

Если внутри квадратных скобок стоит одно двоеточие, то считается, что это двоеточие между первым и вторым параметрами ( $n$  и  $m$ ). В этом случае сечение строки — подпоследовательность исходной строки начиная с символа с индексом  $n$  и заканчивая символом с индексом  $m-1$ .

- Если не указан первый параметр (начало подстроки), то его значение полагается равным 0
- Если не указан второй параметр (конец подстроки), то он полагается равным `len(s)`
- Если не указан третий параметр (шаг), то он полагается равным 1.

Примеры:

↓

```
s = 'abcdefghijklmnopqrstuvwxy'
s2 = '_'
i = 10
print(s[:])      # 'abcdefghijklmnopqrstuvwxy'
print(s[20:])    # 'vwxyz'
print(s[:10])    # 'abcdefghij'
print(s[::2])    # 'acegikmoqsuw'
print(s[10:13:3]) # 'k'
print(s[::-1])   # 'zyxwvutsrqponmlkjihgfedcba'
print(s[::-2])   # 'zxvtrpnljhfdb'
print(s[5:5])    # ''
print(s[:i] + chr(ord(s[i]) - 32) + s[i+1:]) # 'abcdefghijklmnopqrstuvwxy'
s = s[:i] + s2 + s[i+1:] # изменить символ в строке можно только
                        # заново "собрать" новую строку
print(s)         # 'abcdefghijklmnopqrstuvwxy_'
```

В предпоследнем примере используется тот факт, что в ASCII-таблице начало блока прописных латинских букв и начало блока строчных латинских букв разделяет 32 символа (прописные идут раньше). Так что для получения из любой прописной буквы любой строчной и наоборот достаточно вычесть (или прибавить) к коду символа число 32.

### 3.3 Оператор цикла for

В Python кроме цикла `while` существует ещё одна конструкция, позволяющая многократно повторять один и тот же набор операций. Цикл `for` можно записать разными способами, сейчас нам интересен тот вариант, который позволяет перебирать элементы упорядоченных последовательностей, в частности строк.

Синтаксис его таков:

```
for <ch> in <str>:
    <составной оператор>
```

где `<ch>` — имя переменной (которая может не иметь никакого значения к началу выполнения цикла `for`), `<str>` — выражение строкового типа, `<составной оператор>` — один или более операторов.

Цикл выполняется следующим образом: вначале вычисляется значение выражения `<str>`, если его длина больше 0, то символы этого значения перебираются по одному и для каждого значения выполняется тело цикла. При этом переменная `<ch>` принимает последовательно значения всех символов строки `<str>`.

Если в теле цикла нет оператора `break` и не произойдёт никаких ошибок на этапе выполнения, то цикл гарантированно выполнится ровно `len(str)` раз. Несколько примеров:

↓

```
# подсчёт букв латинского алфавита в строке
s = input('enter s:')
count = 0
for c in s:
    if 'a' <= c <= 'z' or 'A' <= c <= 'Z':
        count += 1
print(count)

# функция without_digits заменяет на пробелы все цифры, входящие в строку
def without_digits(s):
    res = ''
    for c in s:
        if '0' <= c <= '9':
            res = res + ' '
        else:
            res = res + c
    return res
print(without_digits('A1B2C3D4E5F')) # 'A B C D E F '
```

## Глава 4

# Списки, функция range, оператор цикла for (продолжение)

Списки — это упорядоченный набор значений произвольного типа.

### 4.1 Значения списков

По аналогии со строкой, как последовательностью символов, чьё значение задавалось перечислением этих символов, значение списка тоже можно задать перечислением его элементов. Отдельные элементы списка разделяются запятой, а весь список заключается в квадратные скобки:

↓

```
a = [1, 2, 3, 4]
b = [2.71828, 3.14159, -1]
c = ['Vera', 'Nadezhda', 'Lyubov']
d = [True, False]
f = [1, True, 2.333, 'abcdefgh']

# способы задания пустого списка
z = []
z = list()

# списки могут быть вложенными, т.е. элементом списка может быть другой список
quarters = [['Jan', 'Feb', 'Mar'], ['Apr', 'May', 'Jun'],
            ['Jul', 'Aug', 'Sep'], ['Oct', 'Nov', 'Dec']]
```

### 4.2 Операции со списками

Списки можно считать обобщением строк в том смысле, что списки — это тоже упорядоченные последовательности, но значения списка могут быть произвольного типа. Большинство операций, описанных выше для строк, полностью переносятся на списки:

- индексирование

↓

```
g = [1, 2, 3, 'February', [6, 7, 8]]
print(g[1])           # программа выведет целое число 2
print(g[len(g) - 1]) # программа выведет список [6, 7, 8]
print(g[-2])          # программа выведет строку 'February'
```

- конкатенация, умножение на целое число и функция len()

↓

```
g = [1, 2, 3]
f = [4, 3, 2]
f = f + g
print(f)           # программа выведет список [4, 3, 2, 1, 2, 3]
print(len(f))      # программа выведет число 6
g = g * 2
print(g)           # программа выведет список [1, 2, 3, 1, 2, 3]
```

- операция in

↓

```
g = [1, 2, 3, 4]
print(1 in g)      # программа выведет True
print([1] in g)   # программа выведет False,
                  # т.к. список g не содержит ни одного элемента типа список
print([1, 2] in g) # программа выведет False, аналогично предыдущему примеру
```

- сечения (срезы)

↓

```
t = [4, 6, 8, -1, 4, 5, 9, 12, -7, 0, -3]
print(t[:4]) # [4, 6, 8, -1]
print(t[8:]) # [-7, 0, -3]
print(t[3:5]) # [-1, 4]
print(t[::-2]) # [-3, -7, 9, 4, 8, 4]
```

Важное отличие списков от строк заключается в том, что списки можно изменять — как его отдельные элементы, так и перечень самих элементов (добавлять, удалять элементы из последовательности и т.п.). Ниже указаны некоторые такие операции:

- изменение элемента списка с заданным индексом:

↓

```
g = [1, 2, 3, 4]
g[1] = 6
print(g) # программа выведет [1, 6, 3, 4]

# содержимое списков можно изменять, используя срезы:
t = [1, 2, 3, 4, 5]
t[2:5] = [5, 4, 3] # подмножество элементов списка t со 2-го по 4-й
                  # теперь содержит элементы: [5, 4, 3]

print(t)          # а весь список t: [1, 2, 5, 4, 3]
```

Важно понимать разницу между изменением среза и изменением отдельного значения списка. В первом случае изменяется некоторое множество элементов списка (возможно, состоящее из одного элемента) и в правой части оператора присваивания может быть только последовательность элементов (т.е. или срез какого-то списка или весь список). В приведённых ниже примерах (в комментарии к каждой строке кода) указано значение списка `t` после очередной операции присваивания:

↓

```
t = [1, 2, 3, 4]
r = [5, 6, 7, 8]
t[1:3] = r[1:3] # [1, 6, 7, 4]
t[:2] = r[2:]  # [7, 8, 7, 4]
t[:2] = r[3]   # ошибка: попытка присвоить срезу списка t ОДНО значение,
               # а не последовательность значений
t[1:2] = r[3]  # ошибка: в левой части оператора присваивания по-прежнему срез
               # хоть и состоящий из одного элемента
q = t[0] + r[:2] # ошибка: пытаюсь создать новый список q,
                 # состоящий из нулевого элемента списка t и двух первых элементов
                 # списка r применяем операцию конкатенации
                 # к числу (t[0]) и списку (r[:2])
t[2:3] = r[3:] # [7, 8, 8, 4]
t[1] = 99     # [7, 99, 8, 4]
t[2] = [9, 6, 7] # [7, 99, [9, 6, 7], 4]

t[len(t) - 1] = t[len(t) - 1:] # [7, 99, [9, 6, 7], [4]]
```



- добавление элемента `elem` в конец списка `t`:

```
t.append(elem)
```

Пример:

↓

```
# программа считывает последовательность чисел указанной длины
# и сохраняет элементы в списке
k = input('k:')
t = []
i = 0
while i < k:
    t = input('element ' + str(i) + ':')
    t.append(t)
    i = i + 1
print(t)
```

Разница между конкатенацией и добавлением элемента следующая: в первом случае это создание нового значения и присваивание этого нового значения переменной, во втором случае — это *изменение* текущего значения. Так что конкатенация определена и для строк и для списков, а добавление элемента — только для списков.

- удаление последнего элемента списка `<t>`

```
<t>.pop()
```

Операция `pop()` имеет возвращаемое значение — элемент, который был удалён. Пример использования:

↓

```
# удаление всех нулевых значений из конца списка
t = [3, 4, 5, 0, 3, 2, 0, 0, 0]
k = len(t) - 1
while k >= 0 and t[k] == 0:
    t.pop()
    k -= 1
print(t)
```

Последние две операции несколько отличаются способом записи от изученных до сих пор операций и функций. Такие операции называются методами — это функции, применимые к значениям определённого типа. В частности, `append()` и `pop()` — методы, применимые к спискам. В общем случае синтаксис вызова метода таков:

```
<переменная>.<имя метода>(<перечень аргументов>)
```

Можно считать, что метод — это функция, куда неявно передаётся в качестве аргумента `<переменная>`. Такая функция может изменять значение этой переменной и может возвращать значение (как, например, метод `pop()`) или не возвращать, а только изменять передаваемый аргумент (как метод `append()`). Ещё несколько примеров, иллюстрирующих использование `append()` и `pop()`:

↓

```
t = [1, 3, 4, 5, 6, 7]
t.append(99)
print(t)          # программа выведет [1, 3, 4, 5, 6, 7, 99]
print(t.pop())   # программа выведет 99
print(t.pop())   # программа выведет 7
print(t)          # программа выведет [1, 3, 4, 5, 6]
```

В главе 8 будут перечислены основные функции и методы, встроенные в язык Python.

### 4.3 Присваивание и копирование списков

Следующий фрагмент программы иллюстрирует наиболее сложную для новичков особенность языка Python:

↓

```
a = [1, 2, 3]
print(a) # программа выведет [1, 2, 3]
b = a
a[0] = 99
print(b) # программа выведет [99, 2, 3], хотя изменяли список a
```

Подробнее об этой особенности изменяемых объектов (и списков в частности) написано в Главе 10. Кроме того, подробное изложение содержится в [1], глава «Интерлюдия о динамической типизации», параграфы «Разделяемые ссылки» и в параграфе 10.3 «Разделяемые ссылки и изменяемые объекты».

Здесь лишь приведём два способа сделать копию списка таким образом, чтоб изменения одного не приводили к изменению другого.

↓

```
a = [1, 2, 3]
print(a)      # программа выведет [1, 2, 3]
x = a[:]      # копия x создана срезом, содержащим ВСЕ элементы списка a
y = list(a)   # копия y создана встроенной функцией list()
a[0] = 99     # изменение "оригинала"
print(x)      # программа выведет [1, 2, 3]
print(y)      # программа выведет [1, 2, 3]
```

## 4.4 Функция `range`

Кроме создания списка перечислением его элементов ещё один способ предоставляет функция `range`. Её синтаксис:

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

Параметры `start`, `stop`, `step` — целые числа ( $step \neq 0$ ).

Результат выполнения функции `range` при положительном (отрицательном) значении параметра `step` — последовательность целых чисел, элементов арифметической прогрессии  $\{a_n\}$  с начальным элементом, равным числу `start`, с разностью `step` и таких, что  $start \leq a_i < stop$  ( $stop \leq a_i < start$ ).

Если указаны только два параметра, то считается, что первый — это `start`, а второй — `stop`, параметр `step` равен 1.

Если указан один параметр, считается, что это параметр `stop`, а параметры `start` и `step` по умолчанию принимаются равными 0 и 1 соответственно.

Примеры:

```
range(10)      # последовательность --- [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
range(10, 2)   # последовательность --- [] # т.к. предполагается, что
               # указаны параметры start=10, stop=2, step=1
range(2, 5)    # последовательность --- [2, 3, 4]
range(3, 20, 2) # последовательность --- [3, 5, 7, 9, 11, 13, 15, 17, 19]
range(-5, -2)  # последовательность --- [-5, -4, -3]
range(1)       # последовательность --- [0]
range(3, 4, 2) # результат --- [3]
range(-1, -1, -7) # последовательность --- [-1]
range(10, 7, -4) # последовательность --- []
range(0, 20, 20) # последовательность --- [0]
range(10, 10, 10) # последовательность --- []
range(13, -4)  # последовательность --- []
```

## 4.5 Оператор цикла `for`

Цикл `for`, описанный в параграфе 3.3, работает так же и для списков (и строки и списки являются упорядоченными последовательностями).

```
for <elem> in <list>:
    <составной оператор>
```

Необходимо, однако, учитывать, что на разных итерациях цикла переменная `<elem>` может иметь разный тип (список может содержать значения разных типов).

Используя функцию `range` можно создавать список значений цикла `for` "на лету" и значительно расширить область применения цикла или записывать его короче.

В общем случае цикл `for` работает следующим образом. Сначала вычисляется значение последовательности `<list>`. Если она непуста, то переменной `<elem>` присваивается первое значение из набора (с индексом 0). После выполнения тела цикла переменной присваивается следующее значение из последовательности. Последний раз тело цикла выполняется при значении переменной `<elem>`, равном последнему элементу последовательности, т.е. элементу с номером `len(list)-1`. При помощи функции `range()` можно формулировать цикл `for` в терминах индексов последовательности (строки, списка), а не самих элементов.

Конечно, любой цикл `for` можно переписать с помощью цикла `while` и наоборот (используя условный оператор и оператор `break`). Важно научиться понимать, когда удобнее использовать ту или иную конструкцию цикла.

Цикл `for` применяется как правило в случае, когда заранее известно количество итераций цикла или можно заранее получить элементы списка (или их индексы), значения которых надо перечислить с помощью функции `range()`.

Примеры использования:



```
# вывод всех целых чисел от a до b включительно
a = int(input())
b = int(input())
for i in range(a, b+1):
    print(i, end = ' ')

# напечатать все нечётные положительные числа, не превосходящие n, в убывающем порядке
n = int(input())
for i in range(n-(n+1)%2, 0, -2):
    print(i, end = ' ')

# вычисление суммы квадратов первых n натуральных чисел
p = 0
for i in range(n):
    p += i * i
print(p)

t = []
for i in range(int(input('k:'))):
    t.append(int(input('element '+str(i)+':'))) # ввод элементов списка

# вычисление суммы элементов списка имеющих чётный индекс
s = 0
for i in range(0, len(t), 2): # цикл по чётным числам, не превосходящим длину списка
    s += t[i]
print(s)

# пример с чтением k чисел и созданием списка
k = input('k:')
t = []
for i in range(k):
    t = input('element ' + str(i) + ':')
    t.append(t)
print(t)

# то же самое, только короче
t = []
for i in range(int(input('k:'))):
    t.append(int(input('element ' + str(i) + ':')))
print(t)

## использование двух способов записи цикла: while и for
## вычисление минимального положительного элемента списка
k = 0
# ищем первый положительный элемент
while x[k] <= 0:
    k += 1
mn = x[k]

# среди оставшихся ищем минимальный положительный
for i in range(k+1, len(x)):
    if x[i] > 0 and x[i] < mn:
        mn = x[i]

# изменение параметра цикла внутри тела цикла:
# с началом новой итерации параметр всегда получает значение из range
for i in range(5):
    print(i, end = ' ')
    i += 1
    print(i, end = ', ')
# вывод: 0 1, 1 2, 2 3, 3 4, 4 5,
```

## 4.6 Функция random. Задание элементов списка случайными числами.

Функция `random()` возвращает случайное вещественное число на полуинтервале  $[0.0, 1.0)$ . Функция `random()` не является стандартной функцией языка Python, для её использования необходимо подключить одноимённый модуль `random` и разрешить использование соответствующей функции из него. Делается это такой строчкой:

```
from random import random
```

Эту строчку лучше поместить в первой строчке файла, содержащего программу. Ниже приводится несколько примеров использования функции `random()`:

↓

```
from random import random

t = []
# создаём список из 10 случайных вещественных чисел в полуинтервале [0.0, 1.0)
for i in range(10):
    t.append(random())

# то же самое, только одной строчкой
t = [random() for i in range(10)]

# список из 10 случайных целых чисел в полуинтервале [0, 100)
r = [int(random()*100) for i in range(10)]

# список из 10 случайных чётных чисел, принадлежащих отрезку [0, 8]
r = [int(random() * 5) * 2 for i in range(10)]

# пример простого интерфейса
# выбор варианта заполнения списка (случайный, с клавиатуры)
from random import randint

n = int(input('Enter the list length:'))
s = input('(r)andom, (m)anual:')
while not s in ['m', 'M', 'r', 'R']:
    s = input('(r)andom, (m)anual:')
if s == 'm' or s == 'M':
    print('Enter list values (one string)')
    x = [int(input('x[%d]='%i)) for i in range(n)]
else:
    x = [randint(1, 100) for i in range(n)]
print(x)
```

## 4.7 Операции с вложенными списками

Как, используя списки, представить, например, шахматную доску или поле для игры в крестики-нолики?

Как мы помним, элементами списка могут быть сами списки. Такие списки называются вложенными.

Ниже указан вариант интерпретации поля для игры в крестики-нолики, где пустое поле обозначено нулём, крестик числом 1, а нолик числом  $-1$ . Крестики начинали. Кто выиграл и на каком ходу?

↓

```
field = [[1, 0, -1], [-1, 1, -1], [0, 0, 1]]
```

Каким образом обращаться к элементам вложенных списков?

↓

```
field = [[1, 0, -1], [-1, 1, -1], [0, 0, 1]]
print(field[1]) # элемент списка field с индексом 1 это список [-1, 1, 1]
print(field[1][0]) # а элемент с индексом 0 списка [-1, 1, 1] это число -1
```

Таким образом, если представить, что вложенный список выписывается построчно, то при обращении к конкретному элементу сначала указывается индекс строки, а затем индекс столбца.

Ниже приводятся примеры создания таких списков и их обработки.

- **Считывание вложенного списка**

Вводится последовательность строк, содержащих целые числа, разделённые пробелом:

↓

```
s = input()
t = []
while s != '':
    t1 = [int(x) for x in s.split()]
    t.append(t1)
    s = input()
print(t)
```

Обратите внимание, что в этом фрагменте совершенно неважно сколько строк и столбцов содержится во вводимой таблице. Более того, корректно считается набор с разным количеством чисел в разных строках таблицы, например такой:

```
1 2 3
5
3 4 5 66 -9
-5 -6
```

- **Создание вложенного списка**

Список — квадратная таблица  $10 \times 10$ , заполненная нулями:

↓

```
n = 10
t = [0] * n
x = [t[:] for i in range(n)]
print(x)
# а можно так:
x = [[0 for i in range(10)] for j in range(10)]
print(x)
```

Обратите внимание, что в определении списка `x` участвует `t[:]`, т.е. копия списка `t`, а не сам список `t`. Подробнее о копировании списков см. Главу 10.

- **Заполнение матрицы по шаблону**

Как из матрицы слева получить матрицу справа (на главной диагонали, а также двух диагоналях — выше и ниже главной стоят единицы)?

↓

```
0 0 0 0 0      1 1 0 0 0
0 0 0 0 0      1 1 1 0 0
0 0 0 0 0      0 1 1 1 0
0 0 0 0 0      0 0 1 1 1
0 0 0 0 0      0 0 0 1 1

# предполагаем, что квадратная матрица, заполненная нулями уже дана
# и связана с переменной x
n = 10
t = [0]*n
x = [t[:] for i in range(n)]
print(x)

x[0][0] = 1
for i in range(1, n):
    x[i-1][i], x[i][i], x[i][i-1] = 1, 1, 1

# вывод на экран двумерного списка
for t in x:
    print(' '.join([str(z) for z in t]))
```

- Проверка квадратной матрицы на симметричность

По окончании вложенного цикла результат вычисления — логическая переменная `simm` — имеет значение `True`, если матрица симметричная, и `False` в противном случае.

Во вложенном цикле параметр `j` пробегает значения, начиная с `i+1`, а не с `0` и не `i`.

↓

```
N = len(x[0])
simm = True
for i in range(0,N):
    for j in range(i+1, N):
        if x[i][j] != x[j][i]:
            simm = False
            break
    if not simm:
        break
print(simm)
```

# Глава 5

## Кортежи

Кортежи (`tuple`), как и списки — упорядоченные последовательности элементов произвольной природы. Разница лишь в том, эти последовательности *неизменяемые*. Это означает, что мы не умеем добавлять элементы в кортеж, удалять их оттуда и изменять эти элементы. Впрочем, последнее (изменение элементов) требует уточнения, которое будет дано ниже.

### 5.1 Свойства кортежей

Итак, основные свойства кортежей:

- кортежи — упорядоченные последовательности значений произвольного типа;
- определена операция индексации, т.е. обращение к элементу кортежа указанием индекса (номера) этого элемента в кортеже, причём нумерация, как у всех упорядоченных последовательностей, начинается с нуля;
- кортежи — неизменяемые последовательности, т.е. после создания кортежа нельзя добавлять и удалять его элементы, а также изменять их;
- имеют фиксированную длину и могут иметь любое количество уровней вложенности.

### 5.2 Значения кортежей

Значения кортежей задаются при помощи перечисления через запятую его элементов, взятых в круглые скобки.

↓

```
a = () # пустой кортеж
b = (2, 4) # кортеж из двух элементов
c = (2, 'spam', -1.678) # элементы кортежа могут быть разных типов
d = ('abc') # ! кортеж, состоящий из одного элемента, задаётся не так,
# здесь в правой части оператора присваивания
# на самом деле стоит ОДНА строка 'abc', взятая в скобки
e = ('abc',) # а так правильно
# обязательна запятая после единственного элемента
f = ((3, 4), [5, 'abc']) # вложенный кортеж - его элементами могут быть любые значения
print(f[1], f[0][1]) # [5, 'abc'] 4
```

Требование для кортежа из одного элемента содержать запятую возникает оттого, что круглые скобки (в отличие от, например, квадратных или фигурных) наделены в синтаксисе языка Python смыслом, не связанным с кортежами — они определяют порядок выполнения операций в выражениях. Поэтому выражение, взятое в круглые скобки, воспринимается как «выражение, взятое в круглые скобки», а не как последовательность из одного элемента.

Создание кортежа из последовательности — строки или списка:

↓

```
# описанные ниже операции изменяют объекты --- элементы кортежа,
# потому они запрещены: каждая приводит к ошибке
tpl = ([45, -2, 13, 20], 'list', 4)

tpl[0] = [23, 4]
tpl[1] = 'list example'
tpl[2] = 1 + 3
```

## 5.3 Операции с кортежами

- Конкатенация

```
c = (1, 2) + (3, 4) # создание нового кортежа: c = (1, 2, 3, 4)
```

- Умножение на целое число

```
d = (0, 1) * 3 # d = (0, 1, 0, 1, 0, 1)
```

- Индексирование и срезы

```
d = (1, 3, 5, 7, 9)
t = (d[0], d[2:4]) # t = (1, (5, 7))
```

- \* Изменение объектов, содержащихся в кортеже

Основное отличие кортежей от списков — невозможность изменять кортежи. Под изменением понимается изменение элементов кортежа либо изменение их количества. С изменением количества (добавление, удаление) всё понятно, а вот что такое изменение элементов — надо уточнить.

В языке Python все типы являются изменяемыми или неизменяемыми. К изменяемым типам из пока описанных относится только список (`list`), к неизменяемым — все числовые типы, строки и кортежи.

↓

```
g = tuple('abcd') # результат выполнения: ('a', 'b', 'c', 'd')
g = tuple('abc123') # результат выполнения: ('a', 'b', 'c', '1', '2', '3')
g = tuple([3, 4, 'abc']) # результат выполнения: (3, 4, 'abc')
```

Но элементы кортежей, которые сами являются изменяемыми объектами (например, списками), изменять можно:

↓

```
d = (1, 3, [5, 7])
d[2] = [1, 2] # Ошибка! нельзя заменять элемент кортежа другим элементом
d[2] = [5, 7] # и даже так нельзя: списки хоть и содержат одинаковые элементы,
              # но представляют собой разные объекты (разные области памяти)
d[2][0] = 1 # здесь мы меняем содержимое списка
d[2][1] = 2
print(d) # (1, 3, [1, 2])
```

Подробнее об изменяемых и неизменяемых объектах см. Главу 10, об использовании кортежей в определении функций см. Главу 11.

## 5.4 Применение кортежей

Основные преимущества использования кортежей следующие:

- множественное присваивание: `a, b = b, a`
- кортежи (как неизменяемые объекты, наряду с числами и строками) используются в качестве ключей в словарях (см. главу 6);
- если вы работаете с данными, которые не предполагается изменять в процессе обработки, то использование кортежей автоматически защищает их от случайного изменения (и последующего поиска ошибки). Примером могут служить алгоритмы вычислительной геометрии, работающие с точками как парами их декартовых координат.



# Глава 6

## Словари

В отличие от списков, которые являются *упорядоченными* последовательностями элементов произвольного типа, элементы словаря это неупорядоченные последовательности пар (**ключ: значение**).

Иногда словари называют ассоциативными массивами, иногда отображениями (имеется в виду отображение множества ключей словаря на множество его значений).

Как и списки, словари имеют переменную длину, произвольную вложенность и могут хранить значения произвольных типов.

Из определения словаря следует, что все операции так или иначе использующие определённый порядок следования элементов (индексирование — получение элемента по его номеру, получение среза, конкатенация) для словарей не имеют смысла.

Как и для всех последовательностей, для словаря определена функция `len`, возвращающая количество пар **ключ: значение** в словаре.

### 6.1 Создание словаря

Словарь создаётся при помощи фигурных скобок и перечисления пар **ключ: значение**. В словарях существует ограничение на тип ключей — это могут быть только значения неизменяемого типа: целые числа, строки и кортежи.

↓

```
# пустой словарь
d = {}

# словарь задан перечислением пар ключ:значение
phone = {'Dan': '7926765431', 'John': '74991234567', 'Mary': '79037162534'}

# словарь задан при помощи встроенной функции zip и двух списков: ключей и значений
# функция zip "сшивает" два списка, первый из которых содержит ключи, второй - значения
key_list= ['Dan', 'John', 'Mary', 'Claire']
value_list = ['7926765431', '74991234567', '79037162534', '74957654321']
phone = dict(zip(key_list, value_list))
```

### 6.2 Получение значений словаря, изменение словаря

Создадим словарь:

↓

```
phone = {'Dan': '7926765431', 'Mary': '79037162534'}

print(phone['Dan'])          # 7926765431
print(phone['Mike'])         # Ошибка: KeyError: 'Mike' - такого ключа в словаре нет

if 'Claire' in phone:       # проверка существования ключа Claire в словаре phone
    print(phone['Claire'])
else:
    phone['Claire'] = '74957654321'
print(phone)
# {'Claire': '74957654321', 'Dan': '7926765431', 'Mary': '79037162534'}
# обратите внимание, что порядок пар может не совпадать с заданным при определении
```

Обратите внимание, что оператор вида `словарь[ключ] = значение` имеет разный смысл в зависимости от того, был до этого в словаре такой ключ или нет. В первом случае это изменение значения, соответствующего существующему

ключу, во втором — создание пары **ключ:значение**.

```
# удаление элемента словаря
del phone['John']          # удалены ключ и значение 'John': '74991234567'
```

Однако, если удаляемого ключа нет, такой способ приведёт к ошибке. Удалить ключ можно ещё так (ошибки не возникнет в любом случае):

```
# безопасное удаление элемента словаря
phone.pop('John', None)
```

Оператор `in` позволяет проверить наличие ключа (но не значения), методы `keys` и `values` — получить списки ключей и значений соответственно. Такие списки полезны при необходимости дополнительной обработки элементов словаря (ключей, значений). Например, когда надо перечислить ключи словаря в определённом порядке.

Оператор `in` позволяет весьма сжато записывать цикл по ключам словаря:

↓

```
s = 0
res = {'Dan':5, 'John':4, 'Mary':4, 'Claire':3, 'Maggie':5}
for x in res:
    s += res[x]
print(s / len(res))
```

Другой полезный метод при работе со словарями — метод `get(key, default)`. Смысл его в следующем: при обращении к значению несуществующего ключа можно предусмотреть значение по умолчанию, которое можно использовать.

```
res = {'Dan':5, 'John':4, 'Mary':4, 'Claire':3, 'Maggie':5}
print(res.get('Dan', 2))    # 5
print(res.get('Mike', 2))  # 2
```

## 6.3 Слияние словарей

Для словарей определена операция слияния, которая реализована в методе `update`:

```
phone1 = {'Dan':'7926765431', 'John':'74881234567'}
phone2 = {'Daniel':'7926765431', 'John':'74775566221'}
phone1.update(phone2)

print(phone1)
# {'Daniel': '7926765431', 'John': '74775566221', 'Dan': '7926765431'}
```

Работает он так: все ключи словаря `phone2`, которых нет в словаре `phone1` добавляются туда вместе со своими значениями. Значения общих для обоих словарей ключей заменяются на значения из словаря `phone2`.

Другие примеры использования словарей:

↓

```
# подсчёт количества вхождения разных слов в данной строке
s = input('s:')
d = dict()
for x in s.split():
    if x in d:
        d[x] += 1
    else:
        d[x] = 1
print(d)
```

То же самое можно записать короче, используя метод `get`:

↓

```
s = input('s:')
d = dict()
for x in s.split():
    d[x] = d.get(x, 0) + 1
print(d)

# s:to be or not to be
# {'to': 2, 'or': 1, 'not': 1, 'be': 2}
```

Теперь, если нужно в качестве ключей словаря иметь не слова, а их частоты (количество вхождений), можно «инвертировать» словарь (ключи — частоты, значения — слова, встречающиеся соответствующее количество раз):

↓

```
def invert_dict(d):
    inv = dict()
    for key in d:
        val = d[key]
        if val not in inv:
            inv[val] = [key]
        else:
            inv[val].append(key)
    return inv

s = input('s:') # to be or not to be
d = {}
for x in s.split():
    d[x] = d.get(x, 0) + 1
    # словарь d={'not': 1, 'to': 2, 'or': 1, 'be': 2}
print(invert_dict(d)) # {1: ['not', 'or'], 2: ['to', 'be']}
```

# Глава 7

## Множества

Множество в языке Python — тип данных в точности совпадающий с одноимённым математическим понятием. Как и словарь, множество является неупорядоченной последовательностью элементов. Таким образом, все операции, связанные с порядком следования элементов (индексирование, срезы, конкатенация) к множествам неприменимы.

В языке Python существует две разновидности множеств — изменяемые (`set`) и неизменяемые (`frozenset`). Подробнее о разнице изменяемых и неизменяемых типов см. параграф 10.3.

### 7.1 Создание множества и логические операции с ними

Множества создаются так:

```
A = set()           # пустое множество
A = {}             # так множество не задают, это пустой словарь
B = {2, 5, 8, 3, 1} # множество задано перечислением своих элементов
```

- Проверка на принадлежность элемента множеству

```
x = 3
y = 89
A = {2, 3, 4, -1}
print(x in A) # True
print(y in A) # False
```

- Проверка на отсутствие элемента в множестве

```
x = 3
y = 4
A = {2, 4, -1}
print(x not in A) # True
print(y not in A) # False
```

- Проверка на пустоту пересечения множеств

```
A = {3, 1, 2}
B = {5, 6, 2}
C = set()
print(A.isdisjoint(B)) # False
print(C.isdisjoint(B)) # True
```

- Является ли одно множество подмножеством другого

```
A = {3, 2, 7}
B = {5, 6, 2}
C = {3, 2, 4, 5, 6}
print(B.issubset(C)) # True
print(B <= C) # True
print(C.issuperset(C)) # True
print(C >= A) # False
```

- Является ли одно множество собственным подмножеством другого

```
A = {2, 6, 5}
B = {5, 6, 2}
C = {3, 2, 4, 5, 6}
print(A < B)      # False
print(C > B)      # True
```

### 7.1.1 Операции над множествами, не приводящие к их изменению

- Объединение двух или нескольких множеств

```
A = {2, 5}
B = {5, 6}
C = {3, 2, 4}
print(A.union(B)) # {2, 6, 5}
print(A)          # {2, 5} множество A при этом не меняется!
print(B | A | C)  # {2, 3, 4, 5, 6}
```

- Пересечение двух или нескольких множеств

```
A = {2, 5}
B = {5, 6}
C = {3, 2, 5, 4}
print(A.intersection(C)) # {2, 5}
print(B & A & C)          # {5}
```

- Разность множеств

```
A = {1, 2, 3, 4, 5, 6, 7}
B = {5, 6}
C = {2, 1, 5}
print(A.difference(C)) # {3, 4, 6, 7}
print(A - B - C)       # {3, 4, 7}
```

- Симметрическая разность множеств

```
A = {1, 2, 3, 4, 5, 6, 7}
B = {2, 6}
C = {2, 1, 4, 6, 9, 8}
print(A.symmetric_difference(B)) # {1, 3, 4, 5, 7}
print(A ^ C)                     # {3, 5, 7, 8, 9}
```

### 7.1.2 Изменение множеств

- Добавить элемент к множеству

```
A = {1, 2}
A.add(4)
```

- Удалить элемент из множества

```
A = {2, 4, 6}
A.discard(2)
A.discard(3)
```

Другой вариант: нельзя удалять отсутствующий в множестве элемент

```
A = {1, 3, 5}
A.remove(5)
A.remove(4) # ошибка!
```

- Удалить все элементы множества

```
A = {5, 1, 6}
A.clear()
```

- Извлечь из непустого множества произвольный элемент и вернуть его значение

```
A = {1, 2, 3, 4}
x = A.pop()
```

- Добавить к множеству элементы других множеств

```
A = {5, 1, 6}
B = {2}
C = {3, 5, 6}
D = {8}
A |= B      # A = {1, 2, 5, 6}, аналог: A.update(B)
A |= C | D  # A = {1, 2, 3, 5, 6, 8}, аналог: A.update(C, D)
```

- Оставить в множестве только элементы, содержащиеся в других множествах

```
A = {2, 4, 1, 6}
B = {3, 4, 1, 5, 6}
C = {2, 3, 4, 5, 6, 7}
C &= A & B      # {4, 6}, аналог: C.intersection_update(A, B)
A &= B          # {1, 4, 6}, аналог: A.intersection_update(B)
```

- Удалить из множества элементы, содержащиеся в других множествах

```
A = {1, 2, 3, 4}
B = {5, 6, 1}
C = {4, 5}
A -= B | C     # {2, 3}, аналог: A.difference_update(B, C)
```

- Оставить в множестве элементы, присутствующие в одном из двух, но не обоих множествах

```
A = {1, 2, 3, 4, 5, 6, 7}
B = {2, 4, 8}
C = {2, 4, 5, 9}
D = {1, 4, 6}
A ^= B      # {1, 3, 5, 6, 7, 8}
D ^= C      # {1, 2, 5, 6, 9}
```

## Глава 8

# Встроенные функции и методы

Многие из перечисленных в этой главе функций были описаны выше. Тем не менее, здесь даётся их формальное определение и примеры их использования. Оригинал на английском языке можно найти здесь:

<http://docs.python.org/3/library/functions.html>

Определения некоторых функций и методов для последовательностей различаются для разных типов, например, метод `count` для списков и строк. Такие определения даны отдельно для каждого типа. Некоторые имеют одинаковый смысл, такие, как функции `len`, `sum`, `all`, `any`. Их описания даны в параграфе «Общие функции» раздела «Обработка последовательностей».

Наконец, в параграфе 'Разные' приведены функции, которые трудно отнести к какому-то определённом типу, например, `max`.

### 8.1 Числовые функции

- функция `round(number[, ndigits])`

Округляет вещественное число `number` до `ndigits` цифр после запятой. Если параметр `ndigits` не указан, то он считается равным 0 и возвращаемое значение имеет целый тип. Если `ndigits` отрицательное число, то округление происходит до `|ndigits|` знаков до запятой.

Результат округления — вещественное число, ближайшее кратное числу  $10^{-ndigits}$ .

Если результат округления — целое число, равноотстоящие от числа `number` то результатом будет чётное.

```
print(round(3.476))      # 3
print(round(0.5))       # 0
print(round(-0.5))      # 0
print(round(21235.675, -1)) # 21240.0
print(round(2.675, 2))  # 2.67
```

Почему в последней строке результат равен не 2.68? Дело в том, что двоичное представление числа 2.675 является приближённым и выраженное в десятичной системе счисления равно:

```
2.67499999999999982236431605997495353221893310546875
```

Чтобы узнать истинное представление вещественного числа, можно воспользоваться функцией `Decimal` из модуля `decimal`:

```
print(Decimal(44)) # 44 для целых чисел результат равен самому числу
print(Decimal(3.5)) # 3.5 для некоторых вещественных - тоже
print(Decimal(3.55)) # 3.54999999999999982236431605997495353221893310546875
```

- функция `abs(x)`

Возвращает модуль переданного в качестве параметра числа.

```
print(abs(-3)) # 3
print(abs(0.5)) # 0.5
```

- функция `pow(x, y[, z])`

Возвращает  $x^y$ , а если указан параметр `z`, то  $x^y \bmod z$ . Последнее вычисляется быстрее, чем `pow(x, y) % z`.

Если  $y < 0$ , то результат вычисления всегда вещественное число.

```
print(pow(2, 100)) # 1267650600228229401496703205376
print(pow(1, -0.5)) # 1.0
print(pow(3, 9, 5)) # 3
```

## 8.2 Обработка последовательностей

### 8.2.1 Общие функции

- функция `len(t)`

Возвращает длину последовательности `t`, которая может быть строкой, списком, кортежем, словарём, множеством.

```
print(len('abcd'))           # 4
print(len([]))              # 0
# обратите внимание, как вычисляется len у вложенных последовательностей
print(len((2, 'string', (2, 'tuple')))) # 3
```

- функция `sum(x[, start])`

Возвращает сумму элементов последовательности `x`, сложенную с `start`:

$$start + \sum x_i$$

Если параметр `start` не указан, то он считается равным нулю.

```
print(sum([12, 34, 56]))           # 102
print(sum([12, 34, 56], -100))     # 2, start = -100
# списки тоже умеем складывать, результат - список
print(sum([[1, 1, 1],[1, 0, -1], [0, 1, 1]])) # ошибка! по умолчанию start=0
# списки, содержащиеся в исходном списке
# будут прибавляться к нулю, а для целых чисел и списков
# операция сложения не определена
print(sum([[1, 1, 1],[1, 0, -1], [0, 1, 1]], [])) # [1, 1, 1, 1, 0, -1, 0, 1, 1]
```

- функции `all(x)` и `any(x)`

Функция `all` возвращает логическое значение `True`, если все элементы последовательности равны `True` (или последовательность пуста) и `False` в противном случае.

Напоминание о приведении типов: к значению `True` приводятся все отличные от нуля числа и любые непустые последовательности, к значению `False` приводится число нуль и все пустые последовательности.

```
print(all([]))                 # True (список пуст)
print(all([[]]))              # False (список не пуст, содержит один элемент)
print(all([True, []]))        # False (непустой список,
# одно из значений которого равно False)
print(all([(4.5), True]))     # True
print(all(''), all('ab'))     # True, True
```

Функция `any` возвращает логическое значение `True`, если хотя бы один элемент непустой последовательности равен `True`. В противном случае, а также если последовательность пуста, возвращает `False`.

```
print(any(''), all('ab'))     # False True
```

### 8.2.2 Методы работы со списками

- метод `x.count(a)`

Возвращает количество элементов последовательности `x`, равных `a`.

```
print('kldajfhwfgnwegwegvwnrtgh'.count('h')) # 2
print([[1,2,3], [3], 3].count(3))             # 1
print([[1,2,3], [3], 3].count([3]))           # 1
print(([1,2,3], [3], 3).count([1, 2]))        # 0
```



- метод `x.index(a[, i [, j]])`

Возвращает индекс первого вхождения элемента `a` в срез последовательности `x[i: j]`. Иначе говоря, возвращает наименьшее `k`, такое что  $x[k] = a, i \leq k < j$ . Если такого элемента нет, происходит ошибка.

```
print('abcdefgh'.index('f')) # 5
print('abcdefgh'.index('z')) # ошибка!
print([1, 2, (3,)].index(2)) # 1
print([1, 2, (3,)].index(3)) # ошибка!
print([1, 2, (3,)].index((3,))) # 2
print([1, 2, (3,)].index([1, 2])) # ошибка!
s = 'Happy families are all alike'
print(s.index('i', 7, 20)) # 9
```

- метод `t.append(x)`

Метод добавляет в конец списка `t` элемент `x`. Метод `append` не возвращает значения, только изменяет список `t`.

```
t = [1, 2, 3]
t.append(5)
print(t) # [1, 2, 3, 5]
t.append([9])
print(t) # [1, 2, 3, 5, [9]]
```

- метод `t.insert(i, x)`

Вставляет элемент `x` в `i`-ю позицию списка `t`. Метод `insert` ничего не возвращает.

```
t = [4, 9, 5, 4, -7, 1]
t.insert(3, 200)
print(t) # [4, 9, 5, 200, 4, -7, 1]
```

Действие метода равносильно следующему оператору: `t[i:i] = x`

- метод `t.extend(x)`

Добавляет список `x` в конец списка `t`. Возвращает изменённый список `t`. Действие аналогично операции `t = t + x`, но выполняется быстрее конкатенации.

```
t = [4, 9, 4, -7, 1]
t.extend(5) # ошибка! 5 - число, а не список
t.extend([5, 99])
print(t) # [4, 9, 4, -7, 1, 5, 99]
```

- метод `t.pop(i)`

Метод удаляет `i`-й элемент списка `t` и возвращает этот элемент.

```
t = [7, 8, 3, 0, 7, -6]
x = t.pop(4)
print(x, t) # 7 [7, 8, 3, 0, -6]
x = t.pop()
print(x, t) # -6 [7, 8, 3, 0]
```

Обратите внимание, что параметр `i` метода `pop` является необязательным. Если он не указан, то считается, что удаляется последний элемент.

Если список пуст, произойдёт ошибка.

- оператор `del x`

Действие оператора `del` такое же, как и метода `t.pop(x)`, но в отличие от метода `pop` оператор `del` не возвращает никакого значения.

```
t = [1, 2, 3, 4, 5, 6, 7, 8]
del t[4]
print(t) # [1, 2, 3, 4, 6, 7, 8]
del t[5:]
print(t) # [1, 2, 3, 4, 6]
```

- метод `t.remove(x)`

Удаляет первое вхождение элемента `x` из списка `t`. Аналогично операции `del t[t.index(x)]`

- метод `t.reverse()`

Разворачивает список `t`, меняя порядок следования элементов на противоположный. Не возвращает значения.

```
t = [1, 2, 3]
t.reverse()
print(t)      # [3,2,1]
```

- метод `t.sort([cmp[, key[, reverse]])`

Метод сортирует список `t` в порядке возрастания элементов. Параметры `cmp` и `key` метода `sort` позволяют определять собственный способ сравнения элементов списка, а также, с помощью параметра `reverse`, получать результат в обратном порядке (вместо дополнительного вызова метода `reverse()`). Рассмотрим параметры подробнее:

- параметр `cmp`: имя функции от двух параметров, возвращающей отрицательное число, нуль или положительное число в зависимости от того, является ли первый параметр меньше второго, равен второму, больше второго.

Например, мы можем сортировать целые числа по сумме цифр их десятичной записи.

- параметр `key`: имя функции от одного аргумента, которая вызывается перед тем, как выполнять сравнение. Такой приём удобен при сортировке последовательностей (строк, кортежей или списков), когда надо указать индекс элемента последовательности, по которому надо сортировать сами последовательности.

Например, есть `t` — список кортежей, устроенных следующим образом: (имя, фамилия, средняя оценка, класс). Для сортировки таких кортежей по средней оценке достаточно указать функцию, возвращающую одно значение: 2-й элемент своего аргумента.

- параметр `reverse`: если он равен `True`, то результат сравнений элементов заменяется на обратный.

Ниже приведены результаты сравнения разных способов вызова метода `t.sort()` для списков из `n` кортежей (список `t`) или целых чисел (список `q`). Стоит помнить о том, что использование параметра `key`

↓

```
n = 10
t = [randint(0, 9), randint(0, 9)) for i in range(n)]
q = [randint(0, 99) for i in range(n)]

print(t)
print q

def cmp_point(a, b):
    if a[0] < b[0]:
        return -1
    if a[0] == b[0] and a[1] < b[1]:
        return -1
    if a[1] == b[1]:
        return 0
    return 1

def cmp_sum(a):
    return sum(a)

def cmp_first(x):
    return x[1]

def cmp_digitsum(a):
    s = 0
    while a > 0:
        s += a % 10
        a = a // 10
    return s

t.sort(key = cmp_first) # сортируем кортежи по элементу, имеющему индекс 1
print(t)
t.sort(key = cmp_sum)   # сортируем по сумме элементов кортежа
print(t)
q.sort(key = cmp_digitsum) # сортируем по сумме цифр целого числа
print(q)
```

Результаты вычислений:

```
# исходные данные
[(5, 0), (9, 7), (0, 3), (6, 1), (3, 5), (2, 0), (5, 4), (7, 0), (9, 7), (6, 1)]
[65, 99, 55, 30, 49, 67, 3, 92, 94, 49]

# результаты сортировок
[(5, 0), (2, 0), (7, 0), (6, 1), (6, 1), (0, 3), (5, 4), (3, 5), (9, 7), (9, 7)]
[(2, 0), (0, 3), (5, 0), (7, 0), (6, 1), (6, 1), (3, 5), (5, 4), (9, 7), (9, 7)]
[30, 3, 55, 65, 92, 49, 67, 94, 49, 99]
```

### 8.2.3 Методы работы со строками

- метод `str.capitalize()`

Метод возвращает копию строки `str`, первый символ которой набран в верхнем регистре, а все остальные — в нижнем. Символы, отличные от латинских букв, остаются без изменения.

```
print('family'.capitalize())      # Family
print('english GRAMMAR'.capitalize()) # English grammar
```

- метод `str.center(width[, fillchar])`

Метод возвращает строку `str`, дополненную символами `fillchar` до ширины, равной `width`, а строку `str` помещает посередине.

```
print('abc'.center(4))      # 'abc '
print('abc'.center(2))      # 'abc'
print('abc'.center(11, '_')) # '____abc____'
```

- метод `str.count(sub[, start[, end]])`

Метод возвращает количество неперекрывающихся подстрок `sub` в сечении строки `str[start:end]`.

```
print('abababaca'.count('aba'))      # 2
print('ababababa'.count('ababa'))    # 1
print('eeeeeeee'.count('ee', 3, 7))  # 2
```

- метод `str.endswith(suffix[, start[, end]])`

Метод возвращает значение `True`, если строка `str` заканчивается на строку `suffix` (говорят, что строка `suffix` является суффиксом строки `str`) и `False` в противном случае.

Если указаны параметры `start` и/или `end`, то суффикс ищется в срезе `str[start:end]`.

Если не указан `start`, то суффикс ищется с начала строки, если не указан `end`, то суффикс ищется до конца строки. Если указан только один дополнительный параметр, то считается, что это параметр `start`.

```
print('abababaca'.endswith('aba'))    # False
print('abababaca'.endswith('babaca', 3, 9)) # True
print('aaaabbbb'.endswith('ab', 3))    # False
print('cddccddc'.endswith('cdd', 0, 7)) # True
```

- метод `str.find(sub[, start[, end]])`

Метод возвращает наименьший индекс, с которого подстрока `sub` входит в строку `str`.

Если указаны дополнительные параметры `start` и `end`, то поиск осуществляется в срезе `str[start:end]`. При этом возвращаемый индекс рассчитывается относительно самой строки, а не её среза.

Если строка не найдена, возвращается `-1`.

```
print('this is the end'.find('is ')) # 2
print('this is the end'.find(' is ')) # 4
print('this is the end'.find(' '))   # -1
print('abababaca'.find('bac', 3, 8)) # 5
```

- метод `str.index(sub[, start[, end]])`

Метод аналогичен `find` за тем исключением, что если строка `sub` не найдена, генерируется исключение `ValueError` (об исключениях в этой книге ничего не написано, почитайте самостоятельно).

- метод `str.isalnum()`

Возвращает `True`, если все символы непустой строки — это буквы или цифры, `False` в противном случае.

- метод `str.isalpha()`

Возвращает `True`, если все символы непустой строки — это буквы, `False` в противном случае.

- метод `str.isdigit()`

Возвращает `True`, если все символы непустой строки — это цифры, `False` в противном случае.

- метод `str.islower()`

Возвращает `True`, если все буквы строки — это строчные буквы, `False` в противном случае.

- метод `str.isspace()`

Возвращает `True`, если все символы непустой строки — это пробелы, `False` в противном случае.

- метод `str.isupper()`

Возвращает `True`, если все буквы строки — это прописные буквы, `False` в противном случае.

- метод `str.join(iterable)`

Очень важный и полезный метод!

Возвращает строку, которая получена конкатенацией элементов последовательности (`iterable`), переданной в качестве параметра. Элементы последовательности должны быть строками. Разделитель элементов в данном случае — строка, "от лица" которой был вызван метод. Под строкой с вызовом метода `join` указан результат вывода.

```
print('.'.join('abcde'))
a.b.c.d.e

print(' '.join(['it', 'explains', 'a', 'lot']))
it explains a lot

# если элементы последовательности - не строки, то их следует привести к этому типу
x = [1, 3, 24, 5, 6, 7]
print(' '.join(str(elem) for elem in x))
1 3 24 5 6 7

# или так:
print(' '.join(map(str, x)))

print(', '.join(str(a) for a in (234, 2.3e-3, 'twenty two')))
234, 0.0023, twenty two
```

- метод `str.ljust(width[, fillchar])`

То же, что и `str.center(width[, fillchar])`, но располагает строку `str` не по центру, а "прижимает" её к левому краю.

- метод `str.lower()`

Возвращает строку `str`, все буквы которой переведены в нижний регистр.

```
print('abCDeFGh'.lower()) # abcdefgh
print('123.45'.lower()) # 123.45
print('3E23'.lower()) # 3e23
```

- метод `str.lstrip([chars])`

Возвращает строку, полученную из `str` следующим образом: из начала строки `str` удаляются **все** символы, перечисленные в строке `chars`. Если параметр `chars` не указан, то по умолчанию удаляются пробелы.

```
print(' spacious '.rstrip()) # 'spacious '
print('www.example.com'.rstrip('cmowz.')) # 'example.com'
```

Надо понимать, что в `chars` указывается не префикс, который надо удалить, а именно набор символов. После преобразования *первый* символ получившейся строки гарантированно отсутствует в строке `chars`.

- метод `str.partition(sep)`

Сначала находится место первого вхождения строки `sep` в строку `str`. Метод возвращает кортеж, состоящий из трёх строк: часть до разделителя `sep`, сам разделитель, и часть после разделителя.

Если строка `sep` не найдена, то первый элемент кортежа это исходная строка, остальные два — пустые строки.

```
print('word#text string'.partition('#')) # ('word', '#', 'text string')
```

- метод `str.replace(old, new[, count])`

Возвращает копию строки `str`, в которой все вхождения `old` заменены на `new`. Если указан параметр `count`, то заменяются только первые `count` вхождений.

```
print('13 43 56'.replace(' ', ':'))      # '13:43:56'
print('ddouble dd'.replace('dd', 'd'))   # 'double d'
print('abcabcabc'.replace('bca', 'abc', 1)) # 'aabcabcabc'
```

- метод `str.rjust(width[, fillchar])`

То же, что и `str.center(width[, fillchar])`, но располагает строку `str` не по центру, а "прижимает" её к правому краю.

↓

```
x = [1, 2, 3, 4]
for a in x:
    print(str(a).rjust(4, ' '), end = '')
```

- метод `str.split([sep[, maxsplit]])`

**Очень важный и полезный метод!**

Разбивает строку `str` на подстроки, используя `sep` как разделитель и возвращает их список.

Если параметр `sep` не указан, то считается, что он равен пробелу. Если указан параметр `maxsplit`, то делается не более `maxsplit` разбиений (т.е. список содержит `maxsplit+1` строку). Параметр `sep` может содержать больше одного символа.

Метод по-разному интерпретирует разделитель в зависимости от того — указан ли он явно при вызове или используется значение по умолчанию (пробел):

- если параметр `sep` не указан или указано значение `None`, то группы подряд идущих пробелов интерпретируются, как один разделитель;
- если же параметр `sep` указан явно, то каждый из идущих подряд разделителей интерпретируется, как самостоятельный разделитель и в таком случае в списке появляются пустые строки.

```
print('192.168.0.100'.split('.')) # ['192', '168', '0', '100']
print(' 1 2 3 '.split())          # ['1', '2', '3']
print(' 1 2 3 '.split(' '))      # ['', '1', '', '2', '', '', '3', '', '']
print(' 1 2 3 '.split(None, 1))  # ['1', '2 3 ']
print('1<>2<>3'.split('<>'))      # ['1', '2', '3']
print('1,,2'.split(','))         # ['1', '', '2']
```

Кроме прочего, метод `split` используется в стандартной операции чтения массива целых чисел, записанных в одной строке через пробел.

```
x = list(map(int, input().split()))      # если была введена строка '1 23 34 -43'
print(x)                                 # программа выведет [1, 23, 34, -43]
```

- метод `str.startswith(prefix[, start[, end]])`

Метод возвращает значение `True`, если строка `str` начинается на строку `prefix` (говорят, что строка `prefix` является префиксом строки `str`) и `False` в противном случае.

Если указаны параметры `start` и/или `end`, то префикс ищется в срезе `str[start:end]`.

Если не указан `start`, то префикс ищется с начала строки, если не указан `end`, то префикс ищется до конца строки. Если указан только один дополнительный параметр, то считается, что это параметр `start`.

```
print('abababaca'.startswith('aba'))    # True
print('abababaca'.startswith('babac', 1, 7)) # False
print('aaaabbbb'.startswith('ab', 3))    # True
print('cddccddc'.startswith('ccdd', 0, 7)) # False
```

- метод `str.strip([chars])`

Метод возвращает строку, последовательно обработанную методами `lstrip` и `rstrip`.

- метод `str.upper()`

Возвращает строку `str`, все буквы которой переведены в верхний регистр.

```
print('abCDeFGh'.upper()) # ABCDEFGH
print('123.45'.upper())   # 123.45
print('28mb'.upper())     # 28MB
```

## Глава 9

# Работа с файлами

До сих пор наши программы ничего не сохраняли в результате своей работы. Как правило они работали недолго, входные данные либо создавались самой программой, либо запрашивались у пользователя. Результат работы выводился на экран и следующий запуск программы начинался с чистого листа.

Другие программы могут работать продолжительное время (или постоянно, как операционная система или веб-сервер), могут сохранять результат работы таким образом, чтобы его можно было его использовать после перезапуска программы.

### 9.1 Режимы работы с файлами.

В этой главе понятие текстового файла, пути файла считается известным или по крайней мере интуитивно понятным.

Для работы с файлом нужно его *открыть*. Эту операцию выполняет функция `open(filename, mode)`. Первый параметр (строка) определяет имя открываемого для работы файла и может содержать имя файла (тогда он ищется в текущем каталоге) или его абсолютное имя (вместе с путём, где этот файл находится). Параметр `mode` определяет способ работы с открываемым файлом:

- *чтение* (`mode='r'`): файл только читается, изменять его нельзя
- *добавление* (`mode='a'`): собираемся дописывать информацию в его конец
- *запись* (`mode='w'`): создаём для записи новый файл

Если существующий файл открывается с параметром `'w'`, то его содержимое удаляется.

Если параметр `mode` не указан, то считается, что он равен `'r'`.

Примеры использования:

```
open('text.txt', 'r')           # файл text.txt из текущего каталога
                                # открывается на чтение
open('d:\python\data\a.dat', 'w') # файл a.dat из каталога d:\python\data
                                # открывается для записи
open('\usr\name\file.in', 'a')  # файл file.in из каталога \usr\name
                                # открывается на запись в его конец
```

### 9.2 Чтение из файла

Как было отмечено в предыдущем разделе, работы с файлом начинается с его открытия в одном из трёх режимов. Функция `open` возвращает объект, обращаясь к которому мы и будем выполнять все необходимые манипуляции с файлом. После работы с файлом его необходимо закрыть. В языке Python можно оформить открытие файла в виде отдельного блока, по окончании выполнения которого файл автоматически закроется:

```
with open('test.txt') as f:
    ...
    ...
```

Строчка, начинающаяся со служебного слова `with` является заголовком блока, все операторы которого должны быть записаны ниже со стандартным отступом (как блоки, соответствующие условному оператору или оператору цикла). По окончании блока файл автоматически закрывается.

Для иллюстрации методов работы с файлом был создан файл `alice.txt` с таким текстом:

```
The rabbit-hole went straight on like a tunnel
for some way, and then dipped suddenly down,
so suddenly that Alice had not a moment to think
about stopping herself before she found herself
falling down a very deep well.
```

Для чтения файла в Python используются следующие методы (каждый вызывается для объекта файла, упомянутого выше):

### 1. `f.read(n)`

Параметр `n` — целое число. Если он указан, то метод возвращает строку, состоящую из следующих `n` символов файла. Если параметр не указан, то в строку считывается весь файл целиком, начиная с текущей позиции, т.е. с того символа, до которого файл был прочитан к настоящему моменту.

↓

```
with open('alice.txt') as f:
    print(f.read(10))    # 'The rabbit'
    print(f.read(10))    # '-hole went'
    print(f.read())      # все оставшиеся символы файла
```

### 2. `f.readline()`

Метод читает одну строку (до символа переноса строки `'\n'`). Возвращается прочитанная строка, включая символ переноса строки. Например, ниже показано, как вывести на печать файл с нумерацией его строк:

↓

```
i = 1
with open('alice.txt') as f:
    s = f.readline()
    while s != '':
        print('{0:d} {1:s}'.format(i, s), end = '')
        i += 1
        s = f.readline()
```

Другой пример — чтение из файла `input.txt` двумерной таблицы целых чисел:

↓

```
x = []
with open('input.txt') as f:
    s = f.readline()
    while s != '':
        x.append([int(z) for z in s.split()])
        s = f.readline()

for i in range(len(x)):
    print(' '.join(str(elem) for elem in x[i]))
```

### 3. `f.readlines()`

Метод читает весь файл построчно, возвращает список строк (включая символы переноса строки).

4. Помимо указанных методов работы с файлами в языке Python существует следующая удобная конструкция, позволяющая перебирать строки файла, аналогичная циклу `for` без параметра. Тогда пример с печатью нумерованных строк файла можно записать короче и естественней:

↓

```
i = 1
with open('alice.txt') as f:
    for line in f:
        print('{0:d} {1:s}'.format(i, line), end = '')
        i += 1
```

## 9.3 Запись в файл

Для записи в файл надо открыть файл для записи (режим `w`) или дописывания (режим `a`).

Рассмотрим пример:

↓

```
# создаём в текущем каталоге файл с именем output.txt
# если такой файл уже был, его содержимое стирается
with open('output.txt', 'w') as f:
    # метод write записывает в файл, ассоциированный с переменной f, заданную строку
    f.write('Lewis Carroll')

# открываем файл output.txt для дописывания
# если такого файла не было, он создаётся
with open('output.txt', 'a') as f:
    # первый символ записываемой строки: переход на следующую строку в файле
    f.write("\nAlice's Adventures in Wonderland")
```



## Глава 10

# Динамическая типизация. Переменные, объекты, ссылки.

Мы все время использовали переменные, не объявляя предварительно ни их самих, ни их типы, и все как-то работало. Например, когда выполняется инструкция `a=3`, как интерпретатор Python узнает, что речь идет о целом числе?

Тип переменной в языке Python определяется автоматически во время выполнения программы, а не в результате объявления в программном коде. Поэтому одна и та же переменная в ходе работы программы может

### 10.1 Переменные, объекты и ссылки

Прежде чем мы выясним, как обрабатывается оператор присваивания вроде `a = 3`, разберёмся, как интерпретатор вообще узнаёт о существовании переменных в программе.

#### *Создание переменной*

Переменная (то есть имя) создается автоматически, когда в программе ей впервые присваивается значение. Все последующие операции присваивания просто изменяют значение, ассоциированное с уже созданным именем.

#### *Типы переменных*

Переменные не имеют никакой информации о типе или ограничениях, связанных с ним. Понятие типа присуще объектам, а не именам. Переменные универсальны по своей природе — они всегда являются всего лишь ссылками на конкретные объекты в конкретные моменты времени.

#### *Использование переменной*

Когда переменная участвует в выражении, ее имя замещается объектом, на который она в настоящий момент ссылается, после чего выражение вычисляется. Кроме того, прежде чем переменную можно будет использовать, ей должно быть присвоено значение — использование неинициализированной переменной приведет к ошибке.

Итак:

*Переменная* это ссылка на место в памяти, где хранится объект; переменная создаётся при выполнении первой операции присваивания ей значения; переменная может ссылаться на объект любого типа и ей должно быть присвоено некоторое значение, прежде чем её можно будет использовать.

Это означает, что от вас не требуется заранее объявлять переменные в программе, но вы должны инициализировать их перед использованием — счетчик, например, должен быть инициализирован нулевым значением, прежде чем его можно будет увеличивать.

*Объект* это область памяти, где помимо значения объекта хранится также информация о типе этого значения и счётчик ссылок на это значение.

Таким образом, при выполнении оператора присваивания `a = 3`:

1. Создается объект (выделяется память, записывается значение), представляющий число 3.
2. Создается переменная `a`, если она еще отсутствует.
3. В переменную `a` записывается ссылка на вновь созданный объект, представляющий число 3.

### 10.2 Автоматическая сборка мусора

До сих пор говорилось об особенностях создания и хранения объектов в памяти. А бывают ли ситуации, когда объекты освобождают занимаемую память? Рассмотрим такую последовательность операторов:

```
a = 4
a = 'aaa'
a = 5.9
```

Имя `a` последовательно указывает на объект типа целое число, потом на строку и в конце концов на вещественное число. Но что происходит с прежним значением после того, как выполняется новое присваивание?

Всякий раз, когда имя ассоциируется с новым объектом, интерпретатор Python освобождает память, занимаемую предыдущим объектом (если на него не ссылается какое-либо другое имя). Такое автоматическое освобождение памяти, занимаемой объектами, называется сборкой мусора (*garbage collection*).

В каждом объекте имеется счетчик ссылок, с помощью которого интерпретатор следит за количеством ссылок, указывающих на объект в настоящий момент времени. Как только значение счетчика достигает нуля (и только в этот момент), память, занимаемая объектом, автоматически освобождается. В предыдущем примере мы исходили из предположения, что всякий раз, когда имя `a` ассоциируется с новым объектом, счетчик предыдущего объекта уменьшается до нуля, заставляя интерпретатор освобождать память.

Важно понимать, что в указанном коде *не происходит* создание новой переменной `a`. В памяти создаются *новые объекты* (строка `'aaa'`, вещественное число `5.9`), а переменная `a` всякий раз получает в качестве значения ссылку на новый объект. Прежний объект при этом освобождает память.

Ещё пример:

```
a = 4 # создан объект - целое число 4, на него указывает имя a
b = 4 # в памяти уже существует объект целого типа со значением 4
      # поэтому имя b указывает на него
      # у объекта с целым значением 4 счётчик ссылок теперь равен 2
a = 10 # создаётся объект - целое число со значением 10 (счётчик ссылок равен 1)
       # счётчик ссылок у числа 4 становится равным 1
b = 10 # счётчик ссылок на целое число 10 теперь равен 2 (на него указывают a и b)
       # на 4 не указывает ни одно имя, объект освобождает память
```

### 10.3 Разделяемые ссылки, изменяемые и неизменяемые типы

Рассмотрим другой пример:

```
a = 5
b = a
print(a is b) # True
print(a, b)   # 5 5
```

Оператор `is`, упоминавшийся в Главе 1, отвечает на вопрос — являются ли два объекта идентичными (т.е. занимают одно и то же место в памяти). Идентичные объекты обязательно равны, обратное неверно.

После присваивания `b = a` интерпретатор создаёт переменную `b` и использует для инициализации переменную `a`, при этом она замещается объектом, на который ссылается (5), и `b` превращается в ссылку на этот объект. В результате переменные `a` и `b` ссылаются на один и тот же объект (то есть указывают на одну и ту же область в памяти). В языке Python это называется разделяемая ссылка.

Однако, такие механизмы создания объектов и сопоставления переменных и объектов не являются универсальными. Рассмотрим следующие примеры:

<pre>a = 5 b = 5 print(a is b) print(a, b)</pre>	<pre>a = 5 b = a print(a is b) print(a, b)</pre>	<pre>a = 'abcde' b = 'abcde' print(a is b) print(a, b)</pre>	<pre>a = 'abcde' b = a print(a is b) print(a, b)</pre>	<pre>a = [5, 6] b = [5, 6] print(a == b) print(a is b) a[0] = 9 print(a, b)</pre>	<pre>a = [5, 6] b = a print(a == b) print(a is b) a[0] = 9 print(a, b)</pre>
<pre>True 5 5</pre>	<pre>True 5 5</pre>	<pre>True abcde abcde</pre>	<pre>True abcde abcde</pre>	<pre>True False [9, 6] [5, 6]</pre>	<pre>True True [9, 6] [9, 6]</pre>

Видно, что целый тип и строки ведут себя так, как описывалось в первом примере. Если объект с таким значением уже существует, то его копия не создаётся, а переменной присваивается ссылка на уже существующее значение 5 или `'abcde'`.

Такие типы называются неизменяемыми (*immutable*). Неизменяемыми типами в языке Python являются числа, строки (`str`) и кортежи (`tuple`).

Со списками всё выглядит иначе. Последовательность присваиваний

```
a = [5, 6]
b = [5, 6]
```

очевидно гарантирует нам, что значение выражения `a==b` равно `True`. Но если попытаться выяснить — один и тот же это список (`a is b`), то оказывается, что это не так. Это можно проверить, изменив один из них (`a[0] = 9`).

Семантика же оператора присваивания `b = a` не меняется. После выполнения присваивания переменная `b` указывает на тот же объект, что и `a`. Так что все изменения списка `a` — это и изменения списка `b` — это один и тот же объект.

Такие типы называются изменяемыми (mutable). В языке Python изменяемыми типами являются списки (`list`) и словари (`dict`).

Проверьте вашу интуицию. Что выдаст проверка в таком случае?

```
a = [5, 6]
b = a + []
print(a == b, a is b)
```

Вопрос: как же скопировать список, если нам нужна "независимая" копия? Если список не вложенный, то можно применить три способа:

- сечения:

```
a = [5, 6]
b = a[:]
```

- функция `list()`:

```
a = [5, 6]
b = list(a)
```

- функция `copy` из одноимённого модуля

```
from copy import copy
a = [5, 6]
b = copy(a)
```

Если список вложенный, то ни один из указанных выше методов не даст желаемого результата. Проверьте интуицию снова: что выведет программа в каждом из фрагментов?

```
from copy import copy
a = [[1, 2], [3, 4]]
b = copy(a)
a[0] = [8, 9]
print(a, b)
```

```
from copy import copy
a = [[1, 2], [3, 4]]
b = copy(a)
a[0][0] = 99
print(a, b)
```

Правильно копировать вложенные списки (и словари) надо так:

```
from copy import deepcopy
a = [[1, 2], [3, 4]]
b = deepcopy(a)

# вложенные словари:
a = {'x':[1, 2], 'y': [3, 4]}
b = deepcopy(a)
```

Ещё один пример с вложенными списками:

```
t = [1,2]
x = []
x = [t, t[:]]
print(x)      # [[1, 2], [1, 2]]
t[0] = 99     # изменяем список t, но первый элемент списка x это тот же самый объект!
              # поэтому ничего удивительного в следующем результате:
print(x)      # [[99, 2], [1, 2]]
```

# Глава 11

## Функции, параметры, вызовы функций.

Используя известные к этому времени конструкции (присваивание, ввод/вывод, условный оператор и оператор цикла) можно запрограммировать любой алгоритм. Так и происходит с несложными задачами и небольшими программами. Но с ростом сложности задач оставаться в этих рамках становится всё труднее.

Наряду с основными конструкциями в большинстве языков существуют способы, реализующие идеи *декомпозиции* и *абстракции*.

*Декомпозиция* — способ разбиения текста программы на отдельные блоки, каждый из которых реализует определённую осмысленную задачу, часть общей работы. Одни и те же блоки могут быть использованы в разных местах программы (или нескольких программ).

*Абстракция* — позволяет сконцентрироваться на смысле и сути данных и/или программы, не обращая внимания на особенности реализации этих данных и/или программы.

В языке Python, как и в большинстве других языков, для реализации идей декомпозиции и абстракции используются, в частности, функции. Функции можно рассматривать как способ задавать новые, более сложные «примитивы» (команды) языка.

### 11.1 Определение функции

Синтаксис определения функции:

```
def <имя функции>(<список формальных параметров>):  
    <составной оператор>
```

где <имя функции> — идентификатор, <список формальных параметров> — список идентификаторов, разделённых запятыми, <составной оператор> — один или несколько операторов, которые называются телом функции. По аналогии с условным оператором и оператором цикла тело функции записывается со стандартным отступом. Если список формальных параметров пуст, скобки всё равно ставятся.

Функции можно разделить на два класса:

- функции, возвращающие значение — в теле такой функции должен встретиться хотя бы один оператор `return`.
- функции, не возвращающие значения

Примеры функций, возвращающих значение:

↓

```
# функция возвращает значение: f(x) = x!  
def factorial(x):  
    res = 1  
    if x == 0 or x == 1:  
        return res  
    while x > 1:  
        res = res * x  
        x -= 1  
    return res
```



```
# функция возвращает количество цифр в десятичной записи числа n
def n_length(n):
    length = 0
    while n > 0:
        n //= 10
        length += 1
    return length
```

Оператор `return` выполняет следующие действия:

- вычисляет выражение, стоящее справа от слова `return`
- прекращает выполнение тела функции
- возвращает значение вычисленного выражения в качестве результата работы функции

Таким образом, функции, возвращающие значение, можно использовать в выражениях (в правой части оператора присваивания, условиях операторов (`if`, `while`), оператора `print`).

Тип значения функции определяется типом возвращаемого выражения, стоящего после `return` в теле функции.

## 11.2 Вызов функции, значение `None`

Вызов функции записывается так:

`<имя функции>( <фактические параметры> )`

где `<имя функции>` — имя одной из определённых выше функций, `<фактические параметры>` — список выражений, разделённых запятыми. Перед выполнением функции значения этих выражений будут вычислены. С этими вычисленными значениями будут связаны имена соответствующих формальных параметров в описании функции. Таким образом, количество формальных и фактических параметров должно совпадать.

Примеры вызовов функций, возвращающих значение:

↓

```
# функция возвращает значение: f(x) = x!
def factorial(x):
    res = 1
    if x == 0 or x == 1:
        return res
    while x > 1:
        res = res * x
        x -= 1
    return res

n = int(input('Enter n:'))
k = int(input('Enter k:'))
# вычисление биномиальных коэффициентов
print(factorial(n) // (factorial(k) * factorial(n - k)))

# сокращение дроби
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

a = int(input())
b = int(input())
g = gcd(a, b)
print('{0:d}/{1:d} = {2:d}/{3:d}'.format(a, b, a // g, b // g))

# проверка, является ли число полным квадратом
def is_sqr(n):
    k = 0
    while k * k < n:
        k += 1
    if k * k == n:
        return True
    return False

k = int(input('Enter k:'))
if is_sqr(k):
    print('YES')
else:
    print('NO')
```

Пример вызова функции, не возвращающей значение:

↓

```
# Пример вызова функции, не возвращающей значение:
def step(k):
    i = 1
    while i <= k:
        print('*', end = '')
        i += 1

def stairway(n):
    k = 1
    while k <= n:
        step(k)
        k += 1
        print()

n = int(input('Enter stairway size:'))
stairway(n)
```

Что произойдёт, если попытаться вычислить функцию, не возвращающую значение? Например, поместив её вызов в правую часть оператора присваивания или попытаться напечатать результат её выполнения? В этом случае

функция возвращает значение `None`. Это специфическое значение — единственный представитель одноимённого типа `None`. `None`, приведённое к логическому типу, равно `False`.

↓	↓
<pre>def smart_print(t):     for i in range(len(t)):         print('x[%d] = %d'.format(i, t[i]))  t = [1, 2, 3] smart_print(t)</pre>	<pre>def smart_print(t):     for i in range(len(t)):         print 'x[%d] = %d'%(i, t[i])  t = [1, 2, 3] print(smart_print(t))</pre>
<pre>x[0] = 1 x[1] = 2 x[2] = 3</pre>	<pre>x[0] = 1 x[1] = 2 x[2] = 3 None</pre>

Кому-то может показаться, что печать лишнего слова `None` не является большой проблемой, хотя, например, в автоматических проверяющих системах вывод должен точно соответствовать спецификации. Приведём теперь простой, но не такой безобидный пример.

↓

```
def odd(t):
    if t%2 == 0:
        return False

if odd(int(input())):
    print('odd') # ничего не печатает, т.к. функция вернула None (т.е. False)
else:
    print('even') # всегда печатаем even
```

Функция `odd` по предположению автора должна возвращать значение `True`, если параметр нечётное число и `False` — в противном случае. Видно, что ничего кроме `False` она возвращать не умеет.

Хуже того, при нечётном параметре она вообще ничего не возвращает. "Ничего" в данном случае означает значение `None`, которое при проверке в условном операторе вычисляется как `False` и программа печатает `even` (чётное).

Такие семантические ошибки самые сложные. Дело в том, что на этапе выполнения программы интерпретатором никаких ошибок нет, но иногда программа ведёт себя как положено, иногда — нет. Локализация таких ошибок может оказаться очень непростой задачей.

### 11.3 Запуск программы, содержащей функции

Файл, содержащий текст программы на языке Python выполняется последовательно сверху вниз — оператор за оператором. Определения функций, т.е. операторы, из которых они составлены, — не выполняются. Определение функции лишь описывает последовательность операторов, которую надо выполнить, если функция будет вызвана в тексте программы.

Таким образом, оба приведённых ниже фрагмента выполняются одинаково: тело функции выполняется только в момент её, функции, вызова.

<pre> def factorial(x):     res = 1     if x == 0 or x == 1:         return 1     while x &gt; 1:         res = res * x         x -= 1     return res  n = input('Enter n:') print(factorial(n)) </pre>	<pre> def JustPrint():     print('Hello!')  def factorial(x):     res = 1     if x == 0 or x == 1:         return 1     while x &gt; 1:         res = res * x         x -= 1     return res  n = input('Enter n:') print(factorial(n)) </pre>
---	---

То же справедливо для любого количества функций, определённых в файле с программой на языке Python.

## 11.4 Блоки, области видимости имён, локальные и глобальные переменные

Блоком в Питоне называется последовательность операторов тела функции или всего файла.

В примере, иллюстрирующем функции (см. 11.2), возвращающие параметры, всего 4 блока: 3 блока — функции `factorial`, `n_length` и `is_sqr`, ещё один — включающий в себя первые три (они называются вложенными) и строки после описания функций.

**Областью видимости имени** считается блок, в которой это имя было связано с новым значением, а также все вложенные в него блоки.

Например:

↓

```

def f():
    return k + 1

k = 5
print(f())

```

Здесь областью видимости переменной `k` является весь текст программы вместе с телом функции, т.к. она получила значение в блоке содержащем внутри себя тело функции (а внутри тела функции имя `k` не связывалось ни с каким значением). Таким образом, использование переменной `k` разрешено, программа выведет 6. Такая переменная называется *глобальной*.

Ещё пример:

↓

```

a = 20

def f(x):
    x = x + a
    return x

print(a, f(4))

```

Будут выведены числа 20 и 24. По умолчанию область видимости имени `a` распространяется на функции этого блока, в частности, на функцию `f`.



Другой пример:

↓

```
a = 20

def f(x):
    a = 3
    x = x + a
    return x

print(a, f(4))
```

Будут выведены числа 20 и 7. Внутри функции `f(x)` имя `a` связано со значением 3. Область видимости этой переменной — функция `f`. Таким образом, имя `a` внутри функции существует отдельно от глобального имени `a`. Такая переменная называется *локальной*.

Это же правило действует и для имён формальных параметров. Их имена являются *локальными* переменными:

↓

```
def f(a):
    return a * a

a = 20
print(a, f(4))
```

Программа выведет числа 20 и 16.

А в следующем примере ошибка (`UnboundLocalError: local variable 'a' referenced before assignment`). Связано это с тем, что сам факт упоминания имени `a` в левой части оператора присваивания внутри функции (т.е. изменения значения переменной `a`) делает это имя локальным.

Значит при вычислении значения выражения `a = a + 3` потребуется значение *локальной* переменной `a`, которое пока неизвестно (имя `a` внутри функции не связано ни с каким значением к моменту вычисления выражения `a+3`).

↓

```
a = 20

def f(x):
    a = a + 3
    x = x + a
    return x

print(a, f(4))
```

И даже так нельзя:

↓

```
def f():
    print(a)
    if False:
        a = 0

a = 20
f()
```

Несмотря на то, что оператор присваивания внутри функции *никогда* не будет выполнен, сам факт его присутствия внутри функции делает переменную `a` локальной, а значит при выполнении оператора `print` её значение будет неизвестно.

Исправить ситуацию в первом примере можно так:

↓

```
a = 20

def f(x):
    b = a + 3
    x = x + b
    return x

print(a, f(4))
```

Будут выведены числа 20 и 27.

Подробнее об областях видимости и связывании читать в главе «Execution model» документации на английском языке: <http://docs.python.org/reference/executionmodel.html>

В следующем параграфе будут более подробно рассмотрены способы передачи параметров в функции, и как с этим связано свойство изменяемости (mutability).

## 11.5 Передача переменных изменяемых и неизменяемых типов в качестве параметров функций

Знакомые с языком Паскаль или C++ знают, что существуют два принципиально разных способа передачи параметров функциям: передача параметра по значению и передача параметра по ссылке.

Первый означает, что в функцию передаётся копия и любые изменения этой копии внутри функции никак не отражаются на значении этого параметра снаружи функции.

Второй означает, что в функцию передаётся ссылка на оригинал, поэтому изменение такого параметра внутри функции влечёт изменение оригинала.

Представим себе, что параметр — ТРХ-версия этой книжки, на 23-й странице которой надо исправить опечатку и мы хотим поручить эту операцию корректору. Предположим, что оригинал этого файла находится в сети и доступен каждому желающему.

Чтобы передать такой параметр по значению, можно скопировать файл и передать копию по электронной почте, распечатать книжку или только её 23-ю страницу. Тогда правка останется в копии (в копии файла, на листе бумаги), но никак не отразится на оригинальном файле. Для передачи параметра по ссылке надо сообщить адрес файла корректору, т.е. сообщить, где именно находится оригинал.

В языке Python способ передачи параметра функции зависит от того, каков тип этого параметра: изменяемые типы передаются по ссылке, неизменяемые — по значению.

В приведённых ниже примерах стандартная функция `id()` выводит идентификатор объекта. Можно считать, что это адрес, по которому хранится значение, с которым связана переменная, переданная в качестве параметра функции `id()`.

### 1. ↓

```
def ref_demo(x):
    print(x, id(x)) # локальная для функции переменная-параметр x
                   # связана со значением 10
                   # обе переменные (k и x)
                   # связаны с одним и тем же значением

    x = 42
    print(x, id(x)) # теперь x связана со значением 42
                   # k и x связаны с разными значениями

k = 10
print(k, id(k)) # переменная k связана со значением 10
ref_demo(k)
print(k, id(k)) # целочисленный тип является неизменяемым
                # поэтому все изменения параметра внутри функции
                # означали не изменение, а создание нового объекта
```

Вывод, который мы видим после запуска программы:

```
k=10, id(k)=31041244
x=10, id(x)=31041244
x=42, id(x)=31040860
k=10, id(k)=31041244
```

Конкретные значения функции `id()`, конечно, всякий раз будут другие, но главное — в третьей строчке **всегда** будет отличное от остальных число.

### 2. Допустим, мы хотим описать функцию, осуществляющую параллельный перенос отрезка на заданный вектор.

Отрезок задан координатами своих концов (элемент кортежа с индексом 0 — абсцисса, с индексом 1 — ордината).

Вектор сдвига  $\overrightarrow{OX}$  задан координатами точки  $X$  (точка  $O$  — начало координат).

↓

```
# Такое решение не будет работать просто потому, что кортежи нельзя изменять,
# в первой же строчке функции будет ошибка
def shift(a, b, x):
    a[0], b[0] = a[0]+x[0], b[0]+x[0] # TypeError: 'tuple' object
                                   # does not support item assignment
    a[1], b[1] = a[1]+x[1], b[1]+x[1]

a = (1, 2)
b = (-3, 2)
shift(a, b, (1, 1))
print(a, b)

# В таком решении ошибок не будет, но цели мы не достигнем.
# Передаваемые параметры a и b - это кортежи, поэтому внутри функции shift
# они будут связаны с новыми значениями, которые "останутся" внутри функции
def shift(a, b, x):
    a = (a[0]+x[0], a[1]+x[1]) # создаём кортежи заново
    b = (b[0]+x[0], b[1]+x[1])
a = (1, 2)
b = (-3, 2)
shift(a, b, (1, 1))
print(a, b)          # (1, 2) (-3, 2)
```

Решение задачи может выглядеть так:

↓

```
def shift(a, b, x):
    a = (a[0]+x[0], a[1]+x[1])
    b = (b[0]+x[0], b[1]+x[1])
    return a, b          # если нельзя изменить переданные параметры,
                        # вернём значения, вычисленные внутри функции

a = (1, 2)
b = (-3, 2)
a, b = shift(a, b, (1, 1))
print(a, b)            # (2, 3) (-2, 3)
```

Несколько примеров, иллюстрирующих передачу изменяемых объектов в качестве параметров функции:

<pre>def f1(t):     print(t)     t.append(57)     print(t)  t = [1, 2, 3] f1(t) print(t)</pre>	<pre>def f2(t):     print(t)     t[0] = 99     print(t)  t = [1, 2, 3] f2(t) print(t)</pre>	<pre>def f3(t):     print(t)     t = t + [34,45]     print(t)  t = [1, 2, 3] f3(t) print(t)</pre>	<pre>def f4(t):     print(t)     t = t[:2]+[99]+t[3:]     print(t)  t = [1, 2, 3, 4] f4(t) print(t)</pre>
<pre>[1, 2, 3] [1, 2, 3, 57] [1, 2, 3, 57]</pre>	<pre>[1, 2, 3] [99, 2, 3] [99, 2, 3]</pre>	<pre>[1, 2, 3] [1, 2, 3, 34, 45] [1, 2, 3]</pre>	<pre>[1, 2, 3, 4] [1, 2, 99, 4] [1, 2, 3, 4]</pre>

В каждой функции переменная-параметр `t` — локальная. В начале выполнения функции эта локальная переменная связывается со значением, которое имеет глобальная переменная `t`. Т.е. в начале работы функции на значение передаваемого списка «смотрят» две переменные — одна глобальная, вторая локальная.

Затем в функциях `f1(t)` и `f2(t)` происходит изменение этого значения. В `f1(t)` в список добавляется число 57, в `f2(t)` нулевой элемент этого списка изменяется на 99.

В функциях `f3(t)` и `f4(t)` создаётся *новое значение локальной переменной t*. Значение глобальной переменной `t` при этом не изменяется.

Разберём ещё один пример:

↓

```

def f1(x):          # локальная переменная-параметр x связана
                   # со значением глобальной переменной t, равным [1, 2, 3, 4, 5, 6, 7]
    x = [44] + x[1:] # создано новое значение, равное [44, 2, 3, 4, 5, 6, 7]
                   # с этим значением связана локальная переменная x
    x[1] = 99       # новое значение, созданное внутри функции, изменяется
                   # теперь оно равно [44, 99, 3, 4, 5, 6, 7]

def f2(x):          # локальная переменная-параметр x связана
                   # со значением глобальной переменной t, равным [1, 2, 3, 4, 5, 6, 7]
    x[1] = 99       # изменяется значение, с которым связана локальная переменная x
                   # таким образом, изменяется значение,
                   # с которым связаны две переменных: локальная x и глобальная t
    x = [44] + x[1:] # создано новое значение, равное [44, 2, 3, 4, 5, 6, 7]
                   # с этим значением связана только локальная переменная x
                   # изменённый на предыдущем шаге список сохранил значение
    x[3] = 99       # изменяется новое значение, созданное на предыдущем шаге
                   # значение, связанное с глобальной переменной t

t = [1, 2, 3, 4, 5, 6, 7]
f1(t)
print(t)           # [1, 2, 3, 4, 5, 6, 7]
t = [1, 2, 3, 4, 5, 6, 7]
f2(t)
print(t)           # [1, 99, 3, 4, 5, 6, 7]

```

Важно понимать, что случаи 1 и 2 возможны лишь при передаче в качестве параметров значений изменяемых типов (таких как список). Метод `append` и изменение элемента списка с заданным индексом — суть изменения списка, операции в примерах 3 и 4 — это создание нового списка (в т.ч. при помощи среза существующего списка).

## 11.6 Рекурсивные функции

Рекурсией называется ситуация, когда функция вызывает сама себя (явным образом или косвенно). Классические примеры рекурсии в математике — определение факториала, чисел Фибоначчи.

Знакомые с творчеством Станислава Лема наверняка помнят пример косвенной рекурсии из «Звёздных дневников Ийона Тихого»:

*Нашёл следующие краткие сведения:*

*«СЕПУЛЬКИ — важный элемент цивилизации ардритов с планеты Энтеропия. См. СЕПУЛЬКАРИИ».*

*Я последовал этому совету и прочёл:*

*«СЕПУЛЬКАРИИ — устройства для сепуления (см.)».*

*Я поискал «Сепуление»; там значилось:*

*«СЕПУЛЕНИЕ — занятие ардритов (см.) с планеты Энтеропия (см.). См. СЕПУЛЬКИ».*

При записи рекурсивных функций обычно решаются две задачи:

- определение значений аргумента, при которых рекурсивный вызов не требуется
- переход от вычисления функции при данном значении аргумента к вычислению функции от другого (как правило меньшего) значения.

Примеры рекурсивных решений приведённых ранее задач и некоторых новых:

↓

```
# вычисление n-го числа Фибоначчи: F(1) = F(2) = 1, F(n) = F(n-1) + F(n-2)
def fibonacci(n):
    if n <= 2:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

# вычисление факториала
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n-1)

# алгоритм Евклида вычисления НОД
def gcd(a, b):
    if b == 0:
        return a
    else:
        return gcd(b, a % b)

# проверка строки на палиндром
def IsPalindrom(s):
    if len(s)<=1: return True
    return s[0]==s[len(s)-1] and IsPalindrom(s[1:len(s)-1])

# подсчёт суммы цифр натурального числа
def sum_of_digits(n):
    if n // 10 == 0:
        return n
    else:
        return n % 10 + sum_of_digits(n // 10)
```

Рекурсивные программы почти всегда короче нерекурсивных, но не всегда эффективнее их. В качестве примера можно реализовать рекурсивную и нерекурсивную функции вычисления чисел Фибоначчи и сравнить время их работы при вычислении, например, 50-го числа этой последовательности.

# Литература

- [1] Лутц М. Изучаем Python. М.: Символ, 2011. 1272 с.
- [2] Downey A.B. Think Python. MA.: Green Tea Press, 2008. 234 с.  
<http://www.greenteapress.com/thinkpython/> (дата обращения 24.12.2012)
- [3] Guo P. LEARN programming by visualizing code execution. <http://www.pythontutor.com/visualize.html>  
(дата обращения 24.12.2012)
- [4] Klein B. Python Course. <http://www.python-course.eu/> (дата обращения 24.12.2012)