МИНИСТЕРСТВО ОБРАЗОВАНИЯ СТАВРОПОЛЬСКОГО КРАЯ

Государственное бюджетное образовательное учреждение высшего образования
«Ставропольский государственный педагогический институт»

Шаяхметов Олег Хазиакрамович

Практикум по решению задач на ЭВМ (язык Python)

Учебно-методическое пособие
для бакалавров, обучающихся по направлению подготовки
44.03.05 Педагогическое образование (с двумя профилями подготовки),
профили «Математика» и «Информатика»

Ставрополь
2022

СОДЕРЖАНИЕ

1. **Язык программирования Python** **4**

2. **Типы данных и операции над ними** **10**

3. **Условные операторы** **14**

4. **Циклы while и for** **17**

5. **Сложные условия: and, or, not** **20**

6. **Списки, кортежи и словари** **22**

7. **Циклические алгоритмы** **25**

8. **Вложенные условные операции и циклы** **30**

9. **Функции** **34**

10. **Массивы** **40**

11. **Строки и операции с ними** **45**

12. **Форматирование строк** **51**

13. **Работа с графикой в Python** **53**

14. **Рекурсия** **58**

15. **Алгоритмы обработки массивов** **62**

16. **Сортировка списков** **69**

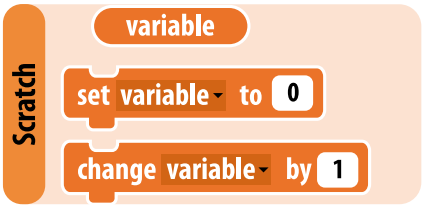
17. **Матрицы** **74**

18. **Приложения** **79**

1. Тема:

Язык программирования Python

Язык программирования Python – это инструмент для создания программ самого разнообразного назначения, доступный даже для новичков. Если вы уже составляли программы из блоков в Scratch, то научиться писать программы на Python для вас будет намного легче. Давайте сравним, как выглядят команды в Scratch и в Python:

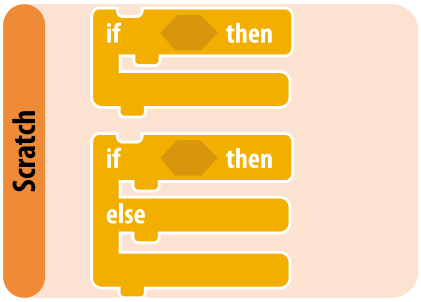



```
variable = 0

variable = variable +1
```

Python

Так выглядят циклы:




```
if True :
    #сделать что-то

if True :
    #сделать что-то
else:
    #сделать что-то
```

Python

Python-код легко читается, а интерактивная оболочка позволяет вводить программы и сразу же получать результат.

На сегодняшний день на этом языке пишутся программы для банков, телекоммуникационных компаний, многие аналитики работают с данными с помощью именно этого языка. Благодаря понятному синтаксису на нем легко начать программировать.

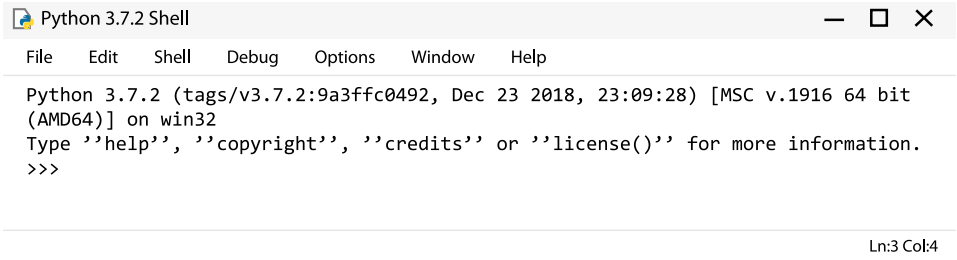
Для программирования на языке Python необходимо установить бесплатную среду программирования у себя на компьютере, которую можно загрузить по адресу: ***https://www.python.org/downloads/.***

Вместе с Python на компьютер установится программа IDLE – среда разработки для написания Python-программ.

Для того чтобы записывать, сохранять и выполнять Python-команды, необходимо сделать следующие шаги:

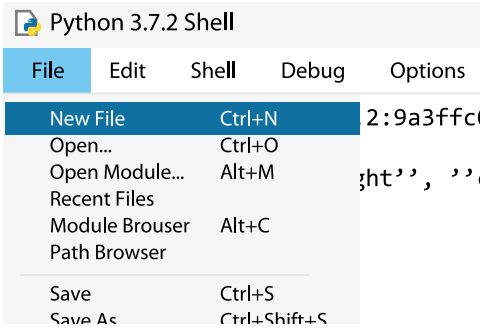
1 Шаг. Запустить IDLE

Откроется окно консоли, в котором вы будете видеть результат вашей программы.



2 Шаг. Создать новый файл

Для записи новой программы необходимо создать отдельный файл. Для этого в меню **File** выберите **New File**:



3 Шаг. Ввести программу

В открывшемся окне введите вашу программу, например:

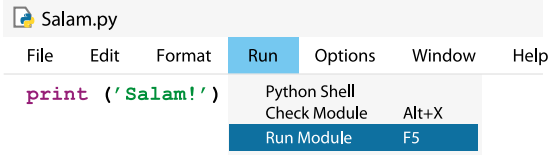
```
print('Salam')
```

4 Шаг. Сохранить программу

В меню **File** выберите **Save As**, задайте новое имя для вашего файла и нажмите **Save**. Мы рекомендуем вам предварительно создать папку My scripts и сохранять туда все свои программы.

5 Шаг. Запустить программу

В меню **Run** выберите **Run Module** (либо можно просто нажать быструю клавишу F5)



Готово! В самой консоли вы должны увидеть сообщение:

```
>>>
Salam!
```

Способ, при котором программа сначала записывается в файл (обычно имеющий расширение **.py**) и при запуске исполняется целиком, называется программным режимом. Такой программный файл на Python называется скриптом (от англ. ***script*** – сценарий).

В Python также бывают пустые программы, которые не содержат ни одного оператора (команды). Часто это комментарии (пишутся после знака #) – пояснения, которые не обрабатываются интерпретатором.

```
#Это пустая программа
```

ОПРЕДЕЛЕНИЯ

Интерпретатор – это программа, которая читает вашу программу и выполняет содержащиеся в ней инструкции.

Функции в Python

В Python есть множество функций, при вызове которых программы выполняют содержащиеся в них команды.

`print()`	`input()`	`randint()`
Функция `print` выводит информацию на экран. Она может выводить как значения переменных, так и значения выражений.	Функция `input`, напротив, позволяет пользователю ввести данные для программы.	Функция `randint` возвращает случайное число из определенного диапазона.

*В скобках указывается значение, с которым будет работать эта функция.

Функция `print` выводит на экран символы, заключенные в апострофы или в кавычки. После выполнения этой команды происходит автоматический переход на новую строку.

Переменные

Напишем алгоритм программы, которая выполняет сложение двух чисел:

1) запрашивает у пользователя два целых числа;
2) складывает их;
3) выводит результат сложения.

ОПРЕДЕЛЕНИЯ

Переменная – это величина, которая имеет имя, тип и значение. Значение переменной может изменяться во время выполнения программы.

В условии появились данные, которые нужно хранить в памяти. Для этого используют переменные. Переменная (как и любая ячейка памяти) может хранить только одно значение. При записи в нее нового значения, предыдущее стирается и его уже никак не восстановить.

В языке Python переменные создаются в памяти при первом использовании, точнее, при присваивании им первоначального значения.

Например, при выполнении оператора присваивания:

```
a = 4
```

в памяти создается новая переменная под именем **a** (объект типа «целое число»). По этому имени теперь можно будет обращаться к переменной: считывать и изменять ее значение.

В именах переменных можно использовать латинские буквы (строчные и заглавные буквы различаются), цифры и знак подчеркивания «_».
Желательно давать переменным соответствующие имена, чтобы можно было сразу понять, какую роль выполняет та или иная переменная.

ЗАПОМНИ

Имя переменой не может начинаться с цифры, иначе транслятору будет сложно различить, где начинается имя, а где – число.

Тип переменной в Python определяется автоматически. Подробнее о типах переменных мы поговорим в следующей теме.

Для написания алгоритма нам нужно решить три подзадачи:

1. ввести два числа с клавиатуры и записать их в переменные (назовем их **a** и **b**);
2. сложить эти два числа и записать результат в третью переменную **c**;
3. вывести значение переменной **c** на экран.

Для ввода значения переменной используется оператор `input`:

```
a = input()
```

При выполнении этой строки система будет ожидать ввода с клавиатуры значения для переменной **a**. При нажатии клавиши *Enter* это значение запишется в переменную **a**. При вызове оператора `input` в скобках можно записать сообщение-подсказку:

```
a = input ('Введите целое число: ')
```

Сложить два значения и записать результат в переменную **c** очень просто:

```
c = a + b
```

Символ «=» – это оператор присваивания, с его помощью изменяют значение переменной. Он выполняется следующим образом: вычисляется выражение справа от символа «=», а затем результат записывается в переменную, записанную слева. Поэтому, например, оператор

`i = i + 1` , увеличивает значение переменной **i** на 1.

Вывести значение переменной **c** на экран оператором `print`:

```
print (c)
```

Вся программа:

Программа на Python	Результат вывода на экран

`a = input ('Введите целое число:')`
`b = input ('Введите целое число:')`
`c = a + b`
`print (c)`
	Введите целое число: 2
	Введите целое число: 3
	23

Мы видим, что два числа не сложились: программа просто объединила их, приписав вторую строку в конец первой. Потому что при такой записи введенные данные воспринимаются оператором `input` как символы, а не числа.

Чтобы исправить эту ошибку, нужно преобразовать символьную строку, которая получена при вводе, в целое число. Это делается с помощью функции **int** (от англ. *integer* – целый):

```
a = int (input())
b = int (input())
```

Возможен еще один вариант: оба числа вводятся не на разных строках, а в одной строке через пробел. В этом случае ввод выполняется сложнее:

```
a, b = map (int, input().split())
```

map() – применяет функцию ко всем элементам списка; в нашем случае это функция **int()**, которая превращает строку в целое число;
split() – строка разрезается на части по пробелам; в результате получается набор значений (список).

В результате после работы функции **map** мы получаем новый список, состоящий уже из чисел. Первое введенное число (первый элемент списка) записывается в переменную **a**, а второе – в переменную **b**.

Запишем программу с учетом рассмотренных функций:

Программа на Python	Результат вывода на экран

`a, b = map (int, input().split())`
`c = a + b`
`print (c)`
	`2 3`
	`5`

Теперь программа работает правильно – складывает два числа, введенных с клавиатуры. Однако у нее есть два недостатка:

1) перед вводом данных пользователь не знает, что от него требуется (сколько чисел нужно вводить и каких);
2) результат выдается в виде числа, которое неизвестно что означает.

Для того чтобы построить диалог программы с пользователем, программу можно записать так:

Программа на Python	Результат вывода на экран

`print ('Введите два целых числа:')`
`a,b = map(int, input().split())`
`c = a + b`
`print (a,'+',b,'=',c,sep='')`
	Введите два целых числа:
	`2 3`
	`2+3=5`

Подсказку для ввода вы можете сделать самостоятельно.
При выводе результата нужно вывести значения трех переменных и два символа: «+» и «=», которые разделяются в операторе `print`**:**

```
print (a, '+', b, '=', c)
```

Для того чтобы убрать лишние пробелы, в операторе `print` используется так называемый разделитель (или сепаратор, англ. *separator*) – **sep**.

```
print (a, '+', b, '=', c, sep = '')
```

Здесь мы установили пустой разделитель (пустую строку). В качестве разделителя можно указать любой знак. К примеру, если указать sep = `'*'` в команде:

```
print (1, 2, 3, 4, sep ='*')
```

то на экране мы увидим: `1*2*3*4*`

КОМПЬЮТЕРНЫЙ ПРАКТИКУМ:

Дан интервал времени в часах, минутах и секундах. Напишите программу, которая определит тот же интервал в секундах.

2. Тема:

Типы данных и операции над ними

Прежде чем использовать переменные, давайте разберемся с типами данных. Набор допустимых операций зависит от выбранного вами типа.

Типы данных

Перечислим основные типы данных в языке Python:

`int` – целые значения;
`float` – вещественные значения (числа с дробной частью);
`bool` – логические значения, **True** (истина, «да») или **False** (ложь, «нет»);
`str` – символ или символьная строка, то есть цепочка символов.

Целые переменные в Python могут быть сколько угодно большими (или, наоборот, маленькими, если речь идет об отрицательных числах): интерпретатор автоматически выделяет область памяти такого размера, который необходим для сохранения результата вычислений. Поэтому в Python легко точно выполнять вычисления с большим количеством значащих цифр.

При записи вещественных чисел в программе, целую и дробную часть разделяют не запятой, а точкой. Например:

```
x = 123.456
```

Логические переменные относятся к типу `bool` и принимают значения `True` (истина) или `False` (ложь).

Программа на Python	Результат вывода на экран

`a = 10`
`b = 3`
`c = a/b`
`print ('c =', float (c))`
	c = 3.3333333333333335

`a = 10`
`b = 3`
`c = a/b`
`print ('c =', int (c))`
	c = 3

`a = 10`
`b = 3`
`c = a/b`
`print ('c =', bool (c))`
	c = True

Арифметические выражения и операции

В Python можно выполнять любые арифметические операции. Арифметические выражения записываются в строчку. Они могут содержать числа, имена переменных, знаки арифметических операций, круглые скобки (для изменения порядка действий) и вызовы функций. Например:

```
a = (c + 5 - 1) / 2 * d
```

При определении порядка действий используется приоритет (старшинство) операций. Они выполняются в следующем порядке:

1. действия в скобках;
2. возведение в степень (**), слева направо;
3. умножение (*) и деление (/), слева направо;
4. сложение и вычитание, слева направо.

Равносильные записи выражений

```
a = b = 0
```

```
b = 0
a = b
```

```
a += b
a -= b
a *= b
a /= b
```

```
a = a + b
a = a - b
a = a * b
a = a / b
```

Нужно помнить, что результат деления (операции «/») – это вещественное число, даже если делимое и делитель – целые и делятся друг на друга нацело. Чтобы получить целый результат деления целых чисел, используют оператор «//». Чтобы выделить остаток от деления, используют оператор «%» (они имеют такой же приоритет, как умножение и деление):

Программа на Python

```
d = 85
a = d // 10 #=8
print (a)
b = d % 10  #=5
print (b)
```

Результат вывода на экран

```
8
5
```

Для отрицательных чисел:

```
print (-7 // 2)
print (-7 % 2)
```

```
-4
1
```

С точки зрения теории чисел, остаток – это неотрицательное число, поэтому –7 = (–4)*2+1, то есть частное от деления (–7) на 2 равно –4, а остаток равен 1.
В Python есть операция возведения в степень, которая обозначается двумя звездочками: «**». Например, выражение $y = 2x^2+z3$ запишется так:

```
y = 2*x**2 + z**3
```

Задача 1. Напишите программу для вычисления площади треугольника, если известны его длина основания и высота.

Из геометрии мы знаем, что площадь треугольника равна произведению половины основания треугольника (a) на его высоту (h):

$$S = \frac{1}{2}ah$$

Эту формулу можно также записать так: ***s=(a*h)/2.***

Теперь запишем программу и введем в качестве примера **6 см** как длину основания, а **4 см** как высоту треугольника.

Программа на Python

```
a = float(input('Введите значение основания: '))
h = float(input('Введите значение высоты: '))
s = (a*h)/2
print ('Ответ: s=',s)
```

Результат вывода на экран

```
Введите значение основания: 6
Введите значение высоты: 4
Ответ: s= 12.0
```

Случайные числа

Чтобы получить случайное целое число, сначала загрузим в Python функцию **randint**. Для этого используем команду **import** в окне консоли:

```
>>> from random import randint
>>> randint (1, 10)
7
```

Функция **randint()** выбрало случайное число в диапазоне от первого до второго числа в скобках. В нашем примере она выбрала цифру 7.

Стандартные функции

Большинство стандартных функций языка Python разбиты на группы по назначению, каждая группа записана в отдельный файл, который называется **модулем**. Математические функции собраны в модуле **math**.

Для подключения этого модуля используется команда импорта (загрузки модуля) – `import math`.

Ниже представлен некоторый функционал для работы с числами:

Команда	Выполняемое действие
`int(x)`	приведение вещественного числа x к целому, отбрасывание дробной части
`round(x)`	округление вещественного числа x к ближайшему целому
`ceil(x)`	округление до ближайшего большего числа
`floor(x)`	округление вниз
`abs(x)`	вычисление модуля числа (абсолютной величины)
`sqrt(x)`	квадратный корень числа x
`sin(x)`	синус x (x указывается в радианах)
`cos(x)`	косинус x (x указывается в радианах)
`tan(x)`	тангенс x (x указывается в радианах)

Для обращения к функциям используется точечная запись: указывают имя модуля и затем через точку название функции:

```
print (math.sqrt(x))
```

Программа на Python	Результат вывода на экран
`x = 25` `print (math.sqrt(x))`	5.0

КОМПЬЮТЕРНЫЙ ПРАКТИКУМ:

*1) Дана температура в градусах Цельсия. Напишите программу для вычисления соответствующей температуры в градусах Фаренгейта. Формула перевода – tF = 9/5 * tC + 32*

2) Напишите программу для вычисления значения выражения:

$$\frac{(a+b)*c+d*\frac{e}{f}}{k+p*\frac{b}{a}+g}*\frac{4}{5}$$

3. Тема:

Условные операторы

Возможности, которые мы рассмотрели до этого, позволяют писать ***линейные*** программы, в которых операторы выполняются последовательно друг за другом, и порядок их выполнения не зависит от входных данных.

В большинстве реальных задач порядок действий может несколько изменяться, в зависимости от того, какие данные поступили.

Из курса 6 класса вы помните, что если возникает выбор между двумя вариантами действий, то для записи такого алгоритма используется конструкция ветвления. Ветвление в Python реализуется с помощью условных операторов. В зависимости от значения, условные операторы направляют программу по одному из путей. Например, программа для системы пожарной сигнализации должна выдавать сигнал тревоги, если данные с датчиков показывают повышение температуры или задымленность.

Условный оператор `if` позволяет сначала проверить условие и только потом принять решение об исполнении или не исполнении дальнейших инструкций. Чтобы понять, как работает оператор `if`, рассмотрим типичные задачи на проверку условий и выбор.

Задача 1. Напишем программу, разрешающую доступ только для тех, чей возраст старше 21 года.

```
a = int(input('Введите свой возраст: '))
if a >= 21:
      print('Доступ разрешен')
else:
      print('Доступ запрещен')
```

Начало каждой «ветви» программы обозначается двоеточием «:». Условие в операторе `if` записывается без скобок.

Если условие, записанное после оператора `if` верно (истинно), то затем выполняются все команды (блок команд), которые расположены до другого условного оператора `else`. Если же условие после `if` неверно (ложно), выполняются команды, стоящие после `else`. В нашем примере, если указать возраст – 13, то исполнится инструкция, записанная после `else`: , то есть доступ будет запрещен.

В Python важную роль играют сдвиги операторов относительно левой границы (отступы). Обратите внимание, что слова `if` и `else` начинаются на одном уровне, а все команды внутренних блоков сдвинуты относительно этого уровня вправо на одно и то же расстояние. Для сдвига используют символы табуляции: одно нажатие на клавишу Tab или четыре пробела.

Если необходимо ввести несколько альтернативных условий, то можно использовать дополнительные блоки `elif` (сокращенное от `else – if`), после которого идет блок инструкций.

```
a=int(input('Введите свой возраст: '))
if a >= 21:
      print('Доступ разрешен')
elif a >= 18:
      print('Доступ частично разрешен')
else:
      print('Доступ запрещен')
```

ЗАПОМНИ

В Python в конце заголовков инструкций с условным оператором всегда ставится двоеточие.

Операторы сравнения

Операторы сравнения позволяют нам сравнить между собой два значения, где в результате выводится значение – либо True, либо False.

Математический символ	Оператор Python	Значение	Пример	Результат
<	<	Меньше, чем	1 < 2	True
>	>	Больше, чем	1 > 2	False
⩽	<=	Меньше или равно	1 <= 2	True
⩾	>=	Больше или равно	1 >= 2	False
=	==	Равно	1 == 2	False
≠	! =	Не равно	1 ! = 2	True

Задача 2. Компания проводит опрос общественного мнения и ее интересуют люди в возрасте от 20 до 70 лет включительно. Нужно написать программу, которая запрашивает возраст претендента и выдает ответ: «подходит» он или «не подходит» по этому признаку.

Пусть в переменной `v` записан возраст. Тогда нужный фрагмент программы будет выглядеть так:

```
if v >= 20 and v <= 70:
  print ('подходит')
else:
  print ('не подходит')
```

В языке Python разрешены двойные неравенства, например:

```
if A < B < C:
```

означает то же самое, что и

```
if A < B and B < C:
```

ЗАПОМНИ

= присваивает значение для переменной (если a = b, то a становится тем же, что и b);

== сравнивает два значения (если a==b, то это запрос на сравнение, и программа выдаст результат True или False).

Задача 3. Запишем программу по сохранению файла с использованием условных операторов и операторов сравнения:

```
ans = input('Вы хотите сохранить файл? (да/нет)')
if ans == 'да':
    print('Выберите папку для сохранения')
if ans == 'нет':
    print('Данные будут утеряны для дальнейшего использования')
else:
    print('Ошибка. Такого варианта ответа нет')
```

Результат вывода будет зависеть от того, какой ответ был выбран: «да» или «нет»:

Вариант 1:

Вы хотите сохранить файл? (да/нет) да

Выберите папку для сохранения

Вариант 2:

Вы хотите сохранить файл? (да/нет) нет

Данные будут утеряны для дальнейшего использования

КОМПЬЮТЕРНЫЙ ПРАКТИКУМ:

1) Даны три числа. Напишите программу, которая выводит количество одинаковых чисел в этой цепочке.
2) Даны длины трех отрезков. Напишите программу, которая проверяет, могут ли они являться сторонами треугольника.
3) Дано натуральное число. Определите, будет ли это число: четным, кратным 4.
4) Напишите программу по обмену валюты в сомах, долларах и евро.

4. Тема:

Циклы while и for

Циклы являются такой же важной частью программирования, как условные операторы. С их помощью можно организовать повторение некоторых частей кода. В Python для записи циклов используются 2 вида команд: **`while`** и **`for`**.

Цикл `while`

While переводится с английского как «пока», то есть цикл (блок команд) выполняется до тех пор, пока не выполнится заданное условие. Для этого в начале очередного шага цикла выполняется проверка условия. Поэтому оно называется циклом с предусловием.

Задача 1. Запишем программу для вывода на экран всех целых чисел от 1 до 5.

Программа на Python	Результат вывода на экран

`d = 0`
`while d<5:`
`  d+=1`
`  print (d)`
	1
	2
	3
	4
	5

Проверяется предусловие так: если в начальный момент значение переменной d будет больше или равно 5, цикл не выполнится ни одного раза.

Задача 2. Рассмотрим пример: определите количество цифр в десятичной записи целого положительного числа. Будем предполагать, что исходное число записано в переменную **`n`** целого типа.

Для решения задачи нужно использовать переменную – **счетчик,** значение которой меняется с каждым новым проходом цикла. Для подсчета количества цифр нужно как-то отсекать эти цифры по одной, с начала или с конца, каждый раз увеличивая счетчик. Начальное значение счетчика должно быть равно нулю, так как до выполнения алгоритма еще не найдено ни одной цифры. Чтобы отсечь последнюю цифру, достаточно разделить число нацело на 10. Операции отсечения и увеличения счетчика нужно выполнять столько раз, сколько цифр в числе.
Как только результат очередного деления на 10 будет равен нулю, это и говорит о том, что отброшена последняя оставшаяся цифра. Программа на

Python выглядит так:

```
count = 0
while n > 0:
    n = n // 10
    count += 1
```

Количество проходов цикла будет равно количеству цифр введенного числа, то есть зависит от исходных данных. Если условие в заголовке цикла никогда не нарушится, цикл будет работать бесконечно долго. В этом случае говорят, что «программа зациклилась». Чтобы остановить зацикленную программу, нужно нажать на *Ctrl+C* в окне консоли.

Цикл `for`

Цикл `for` повторяет команды необходимое количество раз. Данная команда позволяет сделать программу компактнее. Рассмотрим предыдущий пример с использованием цикла `for`:

Программа на Python	Результат вывода на экран
`for i in range (5):` `  print (i)`	0 1 2 3 4

Здесь переменная i (ее называют переменной цикла) изменяется в диапазоне (`in range`) от 0 до 5, не включая 5 (то есть от 0 до 4-х включительно). Таким образом, цикл выполняется ровно 5 раз.

Для того чтобы нам получить идентичный ответ (как в случае с `while`), изменим нашу программу:

Равносильные записи выражений		Результат вывода на экран

`d = 0`
`while d < 5 :`
`  d+=1`
`  print ( d )`
	`for i in range(1,6):`
	`  print ( i )`
		1
		2
		3
		4
		5

Задача 2. Выведем степени числа два от 2^1 до 2^{10} (k = степени двойки).

Равносильные записи выражений	

`k = 1`
`while k <= 10 :`
`  print ( 2**k )`
`  k += 1`
	`for k in range(1,11):`
	`  print ( 2**k )`

В первом варианте переменная **k** используется трижды: для присвоения начального значения, в условии цикла и в теле цикла (увеличение на 1).

Во втором варианте переменная **k** задается диапазоном (**range**) из двух чисел – начальным и конечным значением, причем конечное значение ***не входи***т в диапазон.

Шаг изменения переменной цикла по умолчанию равен 1. Если его нужно изменить, указывают третье (необязательное) число в скобках после слова **range** – это нужный шаг. Например, такой цикл выведет только нечетные степени числа 2:

Программа на Python	Результат вывода на экран
`for k in range(1,11,2):` `  print ( 2**k )`	2 8 32 128 512

С каждым шагом цикла переменная цикла может не только увеличиваться, но и уменьшаться. Для этого начальное значение должно быть больше конечного, а шаг – отрицательным. Следующая программа печатает квадраты натуральных чисел от 5 до 1 в порядке убывания:

Программа на Python	Результат вывода на экран
`for k in range(5,0,-1):` `  print ( k**2 )`	25 16 9 4 1

КОМПЬЮТЕРНЫЙ ПРАКТИКУМ:

1) *Напишите программу, которая получает два целых числа A и B и выводит квадраты всех натуральных чисел в интервале от A до B.*
2) *Дано натуральное число. Напишите программу, которая находит сумму его цифр.*
3) *Дан брусок длиной 20 метров. Напишите программу, которая посчитает, какое минимальное целое количество отрезков длиной 1,5 м и 2 м получится из данного бруска.*

5. Тема:

Сложные условия: and, or, not

В программировании важно уметь правильно выстраивать условия. Часто условия бывают сложными, т.е. состоят из нескольких составных условий, соединенных логическими операторами (связками) «И», «ИЛИ» и «НЕ». В Python они записываются английскими словами «and», «or» и «not».

Логический оператор `and` (логическое умножение)

Сложное условие, где составные условия соединены связкой `and`, возвращает `True`, если все выражения равны `True`. Если же хоть одно из выражений ложно, то всё условие является ложным:

```
x = 5
if x < 10 and x % 3 == 0:
    print('True')
else:
    print('False')
```

Ответом здесь будет False, потому что, согласно второй части условия, заданное число 5 не делится на 3 без остатка. Если бы мы записали выражение так: x % 3 == 0 `and` x < 10, то оно также вернуло бы False. Однако второе сравнение x < 10 не выполнялось бы интерпретатором, так как его незачем выполнять. Ведь первое выражение (x % 3 == 0) уже вернуло ложь, которая, в случае оператора `and`, превращает всё выражение в ложь.

Логический оператор `or` (логическое сложение)

Возвращает True, если хотя бы одно из выражений равно True:

```
x = 5
if x < 10 or x % 3 == 0:
    print('True')
else:
    print('False')
```

Ответом здесь будет True, потому что, согласно первой части условия, заданное число 5 меньше 10, хоть и не делится на 3 без остатка. Именно поэтому, если одно из выражений возвращает True, то второе выражение не оценивается, так как оператор `or` в любом случае возвратит True.

Логический оператор `not` (логическое отрицание)

Унарный оператор `not` превращает истину в ложь, а ложь в истину. Унарный он потому, что применяется к одному выражению, стоящему после него, а не справа и слева от него, как в случае бинарных `and` и `or`.

Вариант 1:

```
x = 8
print (not x < 15)

False
```

Вариант 2:

```
x = 8
print (not x > 15)

True
```

Если в одном выражении одновременно используется несколько или даже все логические операторы, приоритет операций следующий:

1) отношения (<, >, <=, >=, ==, !=)
2) not («НЕ»)
3) and («И»)
4) or («ИЛИ»)

Для изменения порядка действий используют круглые скобки. Рассмотрим пример изменения порядка вычислений в случаях, когда возникают скобки:

Вариант 1:

```
a=4
b=6
c=8
result = c==8 or b<a and not a < 7
print (result)
Результат:
True
Выясним почему:
c == 8 or b < a and not a < 7
 True     False          True
                      False
                 False
           True
```

Вариант 2:

```
a=4
b=6
c=8
result= (c==8 or b<a) and not a < 7
print (result)
Результат:
False
Выясним почему:
(c == 8 or b < a) and not a < 7
  True     False           True
       True               False
              False
```

ВОПРОСЫ И ЗАДАНИЯ:

1) С помощью оператора and составьте два сложных логических выражения, одно из которых дает истину, другое – ложь.
2) Аналогично выполните задачу 1, но уже с оператором or.

6. Тема:

Списки, кортежи и словари

Часто нам нужно держать много однообразных данных в одном файле, например, список учеников школы или номера телефонов в справочнике. В Python такие наборы данных можно организовывать в **списки, кортежи и словари**.

Список ***(list)*** – это структура, состоящая из элементов, расположенных в определенном порядке. Каждому элементу соответствует номер (или индекс), по которому к нему можно обратиться. Для создания списка в квадратных скобках ([]) через запятую перечисляются все его элементы. Например, создадим список членов своей семьи:

```
>>> myfamily = ['father', 'mother', 'sister', 'brother']
```

В данном случае наш список будет храниться в переменной myfamily. Когда список создан, можно написать программу для работы с этим списком. К примеру, напишем приветствие для каждого из членов семьи, используя цикл:

Программа на Python	Результат вывода на экран

`myfamily = ['father', 'mother', 'sister', 'brother']`
`for item in myfamily:`
`  print('Hello', item)`
	Hello father
	Hello mother
	Hello sister
	Hello brother

Список может содержать разные типы объектов. В один и тот же список одновременно можно включать строки, числа, объекты других типов данных:

```
objects = [1, 2.6, 'Hello', True]
```

Списки можно складывать, тогда новый список будет содержать элементы из обоих списков:

```
x = [1, 2, 3, 4]
y = [5, 6, 7, 8]
z = x + y
print (z)
```

Результат:

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

Со списками можно делать много разных операций:

x in A	Проверить, содержится ли элемент **x** в списке **A**. Возвращает True или False	`A = [1, 2, 3, 4, 5, 6, 7, 8]`
		`print (2 in A)`
		`Результат:`
		`True`

min(A)	Найти наименьший элемент в списке **A.**	`A = [1, 2, 3, 4, 5, 6, 7, 8]`
		`print (min(A))`
		`Результат:`
		`1`

max(A)	Найти наибольший элемент в списке **A**.	`A = [1, 2, 3, 4, 5, 6, 7, 8]`
		`print (max(A))`
		`Результат:`
		`8`

Кортеж ***(tuple)***, как и список, представляет собой последовательность элементов. Однако хранящиеся в нем элементы нельзя изменять, добавлять или удалять. Для создания кортежа используются круглые скобки, в которые помещаются его значения, разделенные запятыми:

```
user = ('Timur', 23, 1.10.1998)
print(user)
```

В кортежах удобно хранить свойства объектов, например, имя, возраст, дату рождения. Если вдруг кортеж состоит из одного элемента, то после единственного элемента кортежа необходимо поставить запятую:

```
user = ('Tom',)
```

Словари ***(dictionary)*** – это структура данных, в которой каждый элемент вместо индекса имеет уникальный ключ. Элементы словаря можно изменять. Для создания словаря используются фигурные скобки ({}):

```
dictionary = {ключ1:значение1, ключ2:значение2, ...}
```

Создадим словарь под именем myschool:

```
myschool = {'5 klass':'Anara, Kanat, Pavel', '6 klass':
'Chyngyz, Tina, Emil'}
```

В этом словаре в качестве ключей используются названия классов, а в качестве значений – имена тех, кто учится в этих классах.

В словарь можно добавить значение, пометив его новым ключом:

```
myschool['7 klass'] = 'Elena, Ainura, Dastan'
print (myschool)
```

Результат:

```
{'5 klass': 'Anara, Kanat, Pavel', '6 klass': 'Chyngyz,
Tina, Emil', '7 klass':'Elena, Ainura, Dastan'}
```

Чтобы изменить значение элемента, нужно придать его ключу новое значение:

```
myschool = {'5 klass':'Anara, Kanat, Pavel', '6 klass':
'Chyngyz, Tina, Emil'}
myschool['6 klass'] = 'Matvei, Tina, Salima'
print (myschool)
```

Результат:

```
{'5 klass':'Anara, Kanat, Pavel', '6 klass':'Matvei, Tina, Salima'}
```

Используя цикл `for`, можно вывести на экран только ключи словаря:

```
for i in myschool:
    print(i)
```

Результат:
```
5 klass
6 klass
```

Или вывести только значения словаря:

```
for i in myschool:
    print(myschool[i])
```

Результат:
```
Anara, Kanat, Pavel
Chyngyz, Tina, Emil
```

КОМПЬЮТЕРНЫЙ ПРАКТИКУМ:

Создайте словарь «Рыбы», а его элементы разделите на 3 вида: «речные», «озерные» и «морские рыбы». Выведите на экран сначала только ключи, а потом элементы словаря.

7. Тема:

Циклические алгоритмы

Доказано, что любой алгоритм может быть записан с помощью трех алгоритмических конструкций: циклов, условных операторов, линейных алгоритмов.

СМОТРИ ТАКЖЕ
Тема 3.4 *7 класс*
Циклы while и for

Как вы знаете, **цикл** – это многократное выполнение одинаковых действий.

В 7 классе мы уже начали изучать циклы **while** и **for**. Цикл **for** отличается от цикла **while** тем, что используется для повторения каких-то команд заранее известное количество раз. Цикл **while**, напротив, повторяет какое-то действие в тех случаях, когда мы не знаем, сколько повторений данной команды необходимо. При этом нам известно условие, до исполнения которого требуется повторять цикл.

Рассмотрим применение цикла **for** более подробно. Запись цикла for в Python осуществляется по схеме:

При старте цикла значение первого элемента диапазона будет присвоено переменной, с которой будет выполнено обозначенное действие. При втором проходе переменной будет присвоено значение второго элемента диапазона. И так далее, пока со всеми элементами в диапазоне не будет проведено заданное действие. Рассмотрим пример, в котором переменной **letter** каждый раз присваивается новый элемент из строки ***Python***. Команда print выводит на экран каждую букву этой строки по одной:

```
for letter in 'Python':
        print ('Буква:', letter)
>>>
Буква:  P
Буква:  y
Буква:  t
Буква:  h
Буква:  o
Буква:  n
```

В примере ниже с каждым новым проходом значение переменной увеличивается на число из заданного диапазона:

```
f = 12
for i in range(1,6):
    f = f + i
print (f)
>>>
27
```

Цикл «for i in range(1,6)» выполняется пять раз (6 – не включается). На каждом шаге цикла переменная f увеличивается на i. Первоначальное значение f = 12. В цикле значение изменяется:

1 проход: f = 12+1=13
2 проход: f = 13 +2=15
3 проход: f = 15+3=18
4 проход: f = 18+4=22
5 проход: f = 22+5=27

Кратко можно записать так: f = 12 + 1 + 2 + 3 + 4 + 5 = 27

Аргументы для функции range передаются следующим образом:

- **range(x)** – перебирает все значения от 0 до **x**;
  при этом число **x** не включается в диапазон;
- **range(y, x)** – перебирает все значения от **y** до **x**;
  x также не включается в диапазон;
- **range(y, x, s)** – перебирает значения от **y** до **x**, с шагом **s**.

Например:

```
for i in range(0,15,3):
    print(i)
```

В данном случае цикл **for** переберет значения от 0 до 15 с шагом 3, в результате он выведет каждое третье число:

```
>>>
0
3
6
9
12
```

Также в качестве шага можно использовать отрицательные числа, тогда цикл будет перебирать значения в обратном направлении:

```
for i in range(100,0,-10):
    print(i)
>>>
100
80
60
40
20
```

В отличие от цикла **for**, который повторяет последовательность определенное количество раз, цикл **while** руководствуется не количеством, а логическим условием, потому вам не нужно точно знать, сколько раз необходимо выполнить код.
Код цикла **while** будет повторяться до тех пор, пока логическое условие истинно (True).

Задача 1. Давайте попробуем на основе этого цикла написать игру, в которой пользователь должен угадать число, загаданное компьютером. Напишем программу:

```
import random  #загрузим библиотеку случайных чисел
number = random.randint(1, 25) #выбор компьютером случайного числа
choices = 0   #в переменную choices записываем количество попыток
while choices < 5: #выполняет цикл до 5 попыток
    print('Угадай число между 1 and 25:')#предлагает пользо-
вателю ввести число
    guess = input()
    guess = int(guess) #число должно быть целым
    choices = choices + 1 #с каждой попыткой счет увеличивается на 1
    if guess == number:   #если введенное число равно загаданному
        break     #остановить программу
```

Переменной choices присвоено значение 0, которое будет увеличиваться с каждой попыткой угадать число. Мы ограничим программу 5-ю попытками, чтобы программа не попала в бесконечный цикл.

Программа уже работает, но она не сообщает пользователю никаких результатов: пользователь не знает, угадал он число или нет. Результат выглядит так:

```
Угадай число между 1 and 25:
5
Угадай число между 1 and 25:
```

```
16
Угадай число между 1 and 25:
7
Угадай число между 1 and 25:
18
Угадай число между 1 and 25:
10
>>>
```

Для этого введем условные операторы, которые будут сообщать пользователю, что его число меньше или больше загаданного, и это поможет угадать число быстрее:

```
import random
number = random.randint(1, 25)
choices = 0
while choices < 5:
    print('Угадай число между 1 and 25:')
    guess = input()
    guess = int(guess)
    choices = choices + 1
    if guess < number: #если число пользователя меньше загаданного
        print('Мое число больше твоего')
    if guess > number: #если число пользователя больше загаданного
        print('Мое число меньше твоего')
    if guess == number: #если число пользователя равно загаданному
        break
if guess == number:
    print('Молодец! Ты угадал число с' + str(choices) + 'попытки!')
else:
    print('К сожалению, ты не угадал число. Я загадал' + str(number))
```

Если запустить программу, то вариант общения с пользователем будет такой:

```
Угадай число между 1 and 25:
6
Мое число больше твоего
Угадай число между 1 and 25:
17
Мое число меньше твоего
Угадай число между 1 and 25:
```

```
14
Мое число больше твоего
Угадай число между 1 and 25:
15
Молодец! Ты угадал число с 4 попытки!
>>>
```

Теперь программа помогает пользователю угадать число, дает ему подсказки. К примеру, если компьютер загадал число 15, а пользователь ввел 17, программа подскажет, что введенное число больше загаданного.

ВОПРОСЫ И ЗАДАНИЯ:

1) Запишите значение переменной y, полученное в результате работы следующей программы.

```
y = 5
for i in range(2,6):
      y = y + 4 * i
print (y)
```

2) Дан брусок длиной 20 метров. Напишите программу, которая посчитает, какое минимальное целое количество отрезков длиной 1,5 м и 2 м получится из данного бруска.

3) По данному коду запишите в таблицу значения переменных на каждом шаге алгоритма:

k=4 p=1040 m=2

```
while p != m*m:
      k=k+1
      p=p-4
      m=m*2
print (k)
```

k	p	m	m*m

4) У героя Майнкрафта Алекса есть машина, которая выпускает по 4 минерала в минуту. На каждые 100 минералов можно построить новую машину, которая также выпускает по 4 минерала в минуту. Напишите программу, которая вычислит, сколько машин будет у Алекса через час.

8. Тема:

Вложенные условные операции и циклы

В 7 классе мы уже изучали, как работают условные операторы `if` и `else`, которые реализуют в программе две отдельные ветви выполнения. Однако алгоритм программы может предполагать выбор больше, чем из двух путей, например, из трех, четырех или даже больше.

Внутри условного оператора могут находиться любые операторы, в том числе и другие условные операторы. Например, у нас есть показания датчика температуры из печи, требуется определить, высокая она, низкая или в пределах нормы. Нормальной считается температура в диапазоне от 200 до 250 градусов Цельсия. В переменной **t** хранится температура. Один условный оператор не подойдет, потому что есть три возможных результата. Решение задачи можно записать так:

```
t = int (input ('Введите температуру'))
if t > 250:
    print ('Температура в печи очень высокая')
else:
    if 200 <= t <= 250:
        print('Температура в печи в пределах нормы')
    else:
        print('Температура в печи ниже нормы')
```

Условный оператор `if`, проверяющий равенство, находится внутри блока `else (иначе)`, поэтому он называется *вложенным* условным оператором. Как видно из этого примера, использование вложенных условных операторов позволяет выбрать один из *нескольких* вариантов. Если после `else` сразу следует еще один оператор `if`, можно использовать «каскадное» ветвление с ключевыми словами `elif` (сокращение от `else-if`). Рассмотрим задачу, в которой потребуется «каскадное» ветвление: требуется определить четверть координатной плоскости по заданным координатам x и y. В переменных **x** и **y** хранятся целочисленные значения координат, введенные с клавиатуры.

```
x = int(input())
y = int(input())
if x > 0 and y > 0:
```

```
        print('Первая четверть')
    elif x > 0 and y < 0:
        print('Четвертая четверть')
    elif y > 0:
        print('Вторая четверть')
    else:
        print('Третья четверть')
```

В приведенной программе условия `if, ..., elif` проверяются по очереди и выполняется блок, соответствующий первому из истинных условий. Если все проверяемые условия ложны, то выполняется блок `else`, если он присутствует.

Вложенные циклы

В сложных задачах часто бывает так, что на каждом шаге цикла нужно выполнять обработку данных, которая также представляет собой циклический алгоритм. В этом случае получается конструкция «цикл в цикле» или «вложенный цикл».

Цикл называется *вложенным*, если он размещается внутри другого цикла. На первом проходе внешний цикл вызывает внутренний, который исполняется до своего завершения, после чего управление передается в тело внешнего цикла. На втором проходе внешний цикл опять вызывает внутренний. И так до тех пор, пока не завершится внешний цикл.

Задача 1. Выведем на экран таблицу умножения. Для этого во внешнем цикле надо перебрать числа от 1 до 9. Для каждого из них нужно перебрать во внутреннем цикле числа от 1 до 9.

```
1  2  3  4   5  6  7  8  9
2  4  6  8  10 12 14 16 18
```

На одну синюю цифру приходится один ряд черных цифр до 9-ти. Синие 1 и 2 находятся во внешнем цикле, черные цифры во внутреннем.

При этом во внутреннем цикле нужно выполнить умножение переменных счетчиков внешнего и внутреннего цикла первого ряда.

Таким образом, на одно выполнение внешнего цикла произойдет девять выполнений внутреннего, и сформируется одна строка таблицы умножения. После каждой строки надо перейти на новую: это делается во внешнем цикле, после того как закончится выполняться внутренний.

Также для построения таблицы необходимо использовать форматированный вывод, т.е. задавать ширину столбцов (\t), иначе произойдет сдвиг, т.к. количество цифр в каждой строке различно.

Наш код будет выглядеть так:

```
for i in range(1,10):#первый множитель от 1 до 10
    for j in range(1,10):#второй множитель от 1 до 10
        print(i*j, end='\t')
        print()
```

Результат:

```
1   2   3   4   5   6   7   8   9
2   4   6   8   10  12  14  16  18
3   6   9   12  15  18  21  24  27
4   8   12  16  20  24  28  32  36
5   10  15  20  25  30  35  40  45
6   12  18  24  30  36  42  48  54
7   14  21  28  35  42  49  56  63
8   16  24  32  40  48  56  64  72
9   18  27  36  45  54  63  72  81
```

Задача 2. Составим программу нахождения всех простых чисел в интервале от 2 до 100.

Простое число – это число, которое делится без остатка только на 1 и на само на себя. Например, число 5 – простое, так как его можно нацело разделить только на 5 и 1, а число 6 – составное, так как помимо 6 и 1 делится на 2 и 3.

- Единица не является простым числом, поэтому в нашем примере диапазон начинается с цифры 2.
- Согласно условию, если число `n` не имеет делителей в диапазоне от 2 до `n-1`, то оно простое, а если хотя бы один делитель в этом интервале найден, то составное.
- Проверим делимость числа `n` на некоторое число `k`: если остаток равен нулю, то `n` делится на `k`.
- Когда найден хотя бы один делитель, число уже заведомо составное, и искать другие делители в данной задаче не требуется. Для этого при `n%k == 0` выполняется **досрочный выход из цикла с помощью оператора** `break`.
- Переменная flag показывает, есть ли хоть один делитель у числа.

Таким образом, программу можно записать так (здесь `n, k` – целочисленные переменные):

```
for n in range(2,101):
    flag = False
    for k in range(2,n):
        if n % k == 0:
            flag = True
            break
    if not flag:
        print (n)
```

ЗАПОМНИ

Чтобы цикл повторился n раз, последним числом диапазона должно быть n+1

Задача 3. Нарисуем на экране «прямоугольник» из двух видов символов. Края прямоугольника будут отрисованы символом «0», а внутренняя часть символом «1».

Пусть длина прямоугольника будет равна 10 символам, а ширина 7. Внешний цикл, перебирая строки, первой и последней цифрой должен поставить 0. Если строка первая или последняя (поскольку отсчет идет от 0, то это 0-я и 6-я строка), 0 выстраивается от начала и до конца. Во всех остальных случаях – ставим цифру 1. Запишем программу:

```
for i in range(7):                #выводим строку 7 раз
    if i==0 or i==6:              #если строка 1-ая или последняя
        for j in range(20):       #все 20 раз
            print('0',end='')#выводим 0
    else:                         #иначе
        print('0',end='')         #выводим 0
        for j in range(1,19):#кроме 1-ого и 19-ого которые
            print('1',end='')#выводим цифрой 1
        print('0',end='')
    print()
```

ВОПРОСЫ И ЗАДАНИЯ:

1) Напишите программу, которая получает номер месяца и выводит соответствующее ему время года или сообщение об ошибке.
2) Даны ящики, которые вмещают 5 кг, 10 кг и 15 кг яблок. Необходимо выяснить, сколько ящиков разного размера понадобится для того, чтобы распределить 100 кг яблок.

9. Тема:

Функции

Ранее вы уже использовали **встроенные** в интерпретатор функции:

`print()` – *выводит на печать данные, заключенные в круглые скобки;*
`str()` – *преобразует данные к строковому типу;*
`int()` – *преобразует данные к целому числу;*
`float()` – *преобразует целые числа в дробный тип;*
`round()` – *округляет число в большую по модулю сторону.*

Кроме них мы можем создавать свои собственные функции для выполнения тех или иных задач. Для этого в Python предусмотрена возможность, когда некоторый повторяющийся алгоритм (или его фрагмент) оформляется в отдельную функцию.

Для этого новой функции необходимо присвоить имя и описать его алгоритм. В дальнейшем, при упоминании в программе имени функции, запускается соответствующий алгоритм со списком входных и выходных данных. После выполнения функции работа продолжается с той команды, которая непосредственно следует за вызовом функции.
Предположим, что в нескольких местах программы требуется выводить на экран сообщение об ошибке: *«Ошибка в программе»*. Это можно сделать, например, так:

```
print ('Ошибка в программе')
```

Вставка этого оператора вывода везде, где нужно вывести сообщение об ошибке, приведет к загруженности памяти. Если потребуется поменять текст сообщения, то нужно будет искать эти операторы вывода по всей программе. Именно для таких случаев используются функции – вспомогательные алгоритмы, к которым можно обратиться с другого места программы. Запишем функцию `error:`

```
def error():
    print ('Ошибка в программе')
n = int (input())
if n < 0:
    error()
```

Мы ввели новую функцию `error.`

Имя функции начинается с ключевого слова `def` (от англ. *define* – определить), после которого задается уникальное название функции (например, `def sum`). После имени функции ставятся скобки, в которых можно передавать параметры функции и двоеточие. Тело функции записывается с отступом.
Для того чтобы функция заработала в другом месте программы, необходимо ее вызвать по имени (не забыв скобки). Например, **error()**.

Использование функций сокращает код, если какие-то операции выполняются несколько раз в разных местах программы. Иногда большую программу разбивают на несколько функций для удобства и упрощения, оформляя в виде функций отдельные этапы сложного алгоритма. Такой подход делает всю программу более понятной.

Функции и их аргументы

Функциям можно передавать аргументы – дополнительные данные для изменения выполняемых действий.
Предположим, что требуется многократно вывести на экран символ, например, чтобы нарисовать таблицу или разделитель. Программу, решающую эту задачу для переменной n, можно записать так:

```
n = 125 #количество раз
s = '_' #символ
while n > 0:
      print (s, end = '')
      n -= 1;
```

Обратим внимание на вызов функции `print` с именованным аргументом **end** – завершающий символ (по умолчанию – символ «новая строка»).

ЗАПОМНИ

Имена функций должны состоять из строчных букв, а слова разделяться символами подчеркивания – это делает код более удобным для чтения (snake case).

Вспомогательный алгоритм цикла вывода повторяющегося символа можно оформить в виде функции. Этой функции нужно передать *аргументы* – символ и число, сколько раз его повторить. Программа получается такая:

```
def print_char(s, n): #имя функции с аргументами
      k = n
      while k > 0:
            print (s, end = '')
            k -= 1
print_char ('-', 10)  #аргументы
```

Основная программа содержит всего одну команду – вызов функции `print_char`. В скобках указаны **аргументы функции**, указывающие, что символ тире («-») нужно вывести 10 раз.

Глобальные и локальные переменные

Во многих случаях функции используют для обработки данных. Эти данные могут быть глобальными либо локальными. **Локальные переменные** передаются в функцию через аргументы, указанные в круглых скобках после имени функции. Локальные переменные находятся в «зоне видимости» только этой функции и недоступны для всей остальной программы. **Глобальные переменные** доступны во всей программе. К ним можно обратиться по имени и получить связанное с ними значение.

Задача 1. Рассмотрим типы переменных на примере данной программы:

```
def rectangle():
    a = float(input('Ширина: '))
    b = float(input('Высота: '))
    s = a*b
    print('Площадь: ', s)
def triangle():
    a = float(input('Основание: '))
    h = float(input('Высота: '))
    s = 0.5*a*h
    print('Площадь: ', s)
figure = input('1-прямоугольник, 2-треугольник: ')
if figure == '1':
    rectangle()
elif figure == '2':
    triangle()
```

В этой задаче 5 переменных, где глобальная только **figure**. Переменные **a** и **b** из функции rectangle(), и **a** и **h** из функции **triangle()** – локальные. При этом локальные переменные в разных функциях являются разными переменными.

Возврат значений из функции

Каждая функция выдает определенный результат. Если даже вы не указываете на возврат значения в качестве результата, она, тем не менее, выдаст результат None (ничего).

Для того чтобы указать, чему равно значение функции, используют оператор `return` (*англ. вернуть*), после которого записывают значение-результат.
Результатом может быть число, символ, символьная строка или любой другой объект.

Задача 2. Составим функцию, которая вычисляет сумму всех цифр числа (н-р, для числа 147 нужно сложить цифры: 1+4+7=12). Сложение цифр начинаем с последней, в нашем примере это цифра 7.

1. Чтобы получить последнюю цифру числа, нужно взять остаток от деления числа на 10 (147%10=7).
2. Полученный остаток добавляем к «сумме», которая изначально равна нулю (sum=0). Теперь сумма равна 7.
3. Затем мы «отсекаем» последнюю цифру числа 147, используя оператор целочисленного деления (147//10=14).
4. Поскольку цифра 14 > 0, то мы возвращаемся в начало цикла. Цикл продолжается до тех пор, пока значение `n` не становится равно нулю.
5. В нашем примере мы снова отсекаем цифру 4 (14%10= 4) и добавляем ее к сумме (4+7=11).
6. Отделяем ее от всего числа (14//10=1).
7. Последнее однозначное число прибавляем к сумме (11+1).

Результат: 12

Таким образом, наша программа будет выглядеть так:

```
n = int(input('Введите число: '))
def digits_sum (n):
    total = 0
    while n > 0:
        total += n%10
        n = n // 10
    return total
#основная программа
print (digits_sum(n))
```

Задача 3. Напишем программу, которая определяет: делится ли сумма цифр заданного числа на 3. Причем, если в полученной сумме больше одной цифры, то необходимо повторить операцию, пока сумма не будет состоять из единственной цифры.

Например, сумма цифр числа 123456789 равняется 45. Число двузначное – опять находим сумму цифр: 4 + 5 = 9. Получили девятку, которая делится на три (9%3==0). Значит, исходное число 123456789 делится на три. Исходя из этого алгоритма, получим следующую программу:

```
def sum(n):
    sum = 0
    while n>0:
        sum += n % 10
        n = n // 10
    return sum
#основная программа
k = int(input('Введите число: '))
while k > 9:  #пока сумма цифр не будет одной цифрой
    k = sum(k)        #вызываем функцию
if k%3==0:
    print ('Число делится на 3')
else:
    print ('Число не делится на 3')
```

Функцию можно вызывать не только из основной программы, но и из другой функции. Например, функция, которая находит среднее из трех различных чисел (то есть число, заключенное между двумя остальными), может быть определена так:

```
def middle ( a, b, c ):
    mi = min ( a, b, c )
    ma = max ( a, b, c )
    return a + b + c - mi - ma
```

Она использует встроенные функции **min** и **max**. Идея решения состоит в том, что если из суммы трех чисел вычесть минимальное и максимальное, то получится как раз третье число.

Функция может возвращать несколько значений. Например, возврат сразу и частного, и остатка от деления двух чисел можно написать так:

```
def divmod ( x, y ):
    d = x // y
    m = x % y
    return d, m
```

Результат функции можно записать в две различные переменные:

```
a, b = divmod ( 7, 3 )
print ( a, b ) # 2 1
```

Если указать только одну переменную, мы получим кортеж – набор элементов, который заключается в круглые скобки:

```
q = divmod ( 7, 3 )
print ( q ) # (2, 1)
```

В случаях, когда необходимо создать функцию, которая нужна в программе только один раз и выполняет несложную операцию с несколькими аргументами, используют **lambda-функции**. Lambda-функция – это анонимная функция, то есть не имеет своего названия, как в случая с **def**. Запись функции начинается со слова **lambda**, затем через пробел указываются аргументы функции и сразу после двоеточия указывается операция, результат которой будет возвращен функцией. Создадим такую функцию на примере умножения двух чисел:

```
multiple = lambda x, y: x * y #lambda-функция с 2-мя аргументами
print (multiple (2, 5)) #результат 10
```

В примере выше мы присвоили lambda-функцию к переменной **multiple**. Хотя такую функцию можно также записать одной строкой, не присваивая к переменной:

```
print ((lambda x, y: x * y)(2, 6))
```

Одно из главных умений в программировании – это не писать один и тот же код несколько раз. Как только это происходит, нужно понимать, что этот повторяющийся кусок кода должен идти в функцию.

КОМПЬЮТЕРНЫЙ ПРАКТИКУМ:

1) Напишите функцию, которая выводит на экран три переданные ей числа в порядке возрастания.
2) Напишите функцию, которая находит наибольший общий делитель двух натуральных чисел.
3) Напишите функцию, возвращающую знак числа, где аргументом является целое число. Возвращаемый результат 0 – если аргумент равен нулю. -1 – если число отрицательное, 1 – если число положительное.

10. Тема:

Массивы

Основное предназначение современных компьютеров – обработка большого количества данных. При этом надо как-то обращаться к каждой из тысяч (или даже миллионов) ячеек с данными. В этой ситуации имя дают не ячейке, а группе ячеек, в которой каждая ячейка имеет собственный номер. Такая область памяти называется массивом (или таблицей).

Массив – это группа переменных, расположенных в памяти рядом (в соседних ячейках) и имеющих общее имя. Каждая ячейка в массиве имеет уникальный номер (индекс).
Одномерные массивы в Python представляют собой список элементов. Поэтому для работы с массивами используются *списки* (тип данных `list`).

Список в Python – это набор элементов, каждый из которых имеет свой номер (индекс). Нумерация всегда начинается с нуля, второй по счету элемент имеет номер 1 и т.д.

СМОТРИ ТАКЖЕ
Тема 3.2 *8 класс*
Списки, кортежи и словари

Создавать списки можно разными способами. Самый простой способ сделать это – перечислить элементы списка через запятую в квадратных скобках:

```
a = [1, 3, 4, 23, 5]
```

Список – это динамическая структура, его размер можно изменять во время выполнения программы (удалять и добавлять элементы).
Списки можно «складывать» с помощью знака «+», например, показанный выше список можно было построить так:

```
a = [1, 3] + [4, 23] + [5]
```

Сложение одинаковых списков заменяется умножением «*». Вот так создается список из 10 элементов, заполненный нулями:

```
a = [0]*10
```

В более сложных случаях используют генераторы списков – выражения, напоминающие цикл, с помощью которых заполняются элементы вновь созданного списка:

```
a = [i for i in range(10)]
```

Как вы знаете, цикл `for i in range(10)` перебирает все значения i от 0 до 9. Выражение перед словом `for` (в данном случае – `i`) – это то, что записывается в очередной элемент списка для каждого `i`. В приведенном примере список заполняется значениями, которые последовательно принимает переменная `i`, то есть получим такой список:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

То же самое можно получить, если использовать функцию `list` для того, чтобы создать список из данных, которые получаются с помощью функции `range`:

```
a = list (range(10))
```

Для заполнения списка квадратами этих чисел можно использовать такой генератор:

```
a = [i*i for i in range(10)] #[0,1,4,9,16,25,36,49,64,81]
```

В конце записи генератора можно добавить условие отбора. В этом случае в список включаются лишь те из элементов, перебираемых в цикле, которые удовлетворяют этому условию.

Например, следующий генератор составляет список из всех четных чисел в диапазоне от 0 до 9:

```
a = [i for i in range(10) if i%2 ==0] #[0,2,4,6,8]
```

Часто в тестовых и учебных программах списки заполняют случайными числами. Это тоже можно сделать с помощью генератора:

```
from random import randint
a = [randint(20,100) for x in range(10)]
```

Здесь создается список из 10 элементов и заполняется случайными числами из отрезка [20,100]. Для этого используется функция **`randint`**, которая импортируется из модуля **`random`**.

Длина списка (количество элементов в нем) определяется с помощью функции `len`. К примеру:

```
a = [1, 3, 4, 23, 5]
n = len(a)  #5
```

Далее во всех примерах мы будем считать, что в программе создан список `a`, состоящий из `n` элементов (целых чисел). Переменная `i` будет обозначать индекс элемента списка.

Для создания списка из 3-х элементов с использованием цикла запишем следующую программу:

Программа на Python

```
n = 3
a = [0]*3
for i in range(n):
print('a[', i, ']=', sep='',
end="")
a[i] = int(iptum())
```

Результат вывода на экран

```
a[0] = 1
a[1] = 2
a[2] = 3
```

Создать список из n элементов и ввести их значения можно также с помощью генератора списка:

```
a = [int(input()) for i in range(n)]
```

Здесь пользователь должен ввести n-количество элементов, которые преобразуются в целое число с помощью функции **int**, и это число добавляется к списку.

Возможен еще один вариант ввода, когда все элементы списка вводятся в одной строке. В этом случае строку, полученную от функции **input**, нужно «расщепить» на части с помощью метода **split**:

```
data = input()
s = data.split()
print (s)
```

Для вывода списка нужно использовать операцию `print`. Для выведения каждого элемента списка с указанием его индекса в квадратных скобках и по одному элементу в строке используем следующую программу:

Программа на Python

```
a = [1, 2, 3, 4, 5];
for i in range(0, len(a)):
print('a[', i, ']=', a[i],
sep='')
```

Результат вывода на экран

```
a[0] = 1
a[1] = 2
a[2] = 3
a[3] = 4
a[4] = 5
```

Перебор элементов

Перебор элементов списка состоит в том, что мы в цикле просматриваем все элементы списка и, если нужно, выполняем с каждым из них некоторую операцию. Переменная цикла изменяется от 0 до n-1, где n – количество элементов списка:

```
for i in range(0,n):
      a[i] + = 1;
```

в этом примере все элементы списка «a» увеличиваются на 1.

Если сам список изменять не нужно, для перебора его элементов удобнее всего использовать такой цикл:

```
a = [1,2,3,4,5]
for x in a:
      print(x)
```

Здесь вместо `print(x)` можно добавлять любые другие операторы, работающие с копией элемента, записанной в переменную «**x**». Обратите внимание, что изменение переменной «**x**» в теле цикла приведет к ошибке.

Во многих задачах требуется найти некоторые элементы списка, удовлетворяющие заданному условию, и как-то их обработать. Простейшая из таких задач – подсчет нужных элементов. Для решения этой задачи нужно ввести переменную-счетчик, начальное значение которой равно нулю. Далее в цикле просматриваем все элементы списка. Если для очередного элемента выполняется заданное условие, то увеличиваем счетчик на 1.

Предположим, что в списке «**a**» записаны данные о росте детей в классе. Найдем количество учеников, рост которых больше 120 см, но меньше 150 см. В следующей программе используется переменная-счетчик **count**:

```
count = 0
for x in a:
      if (120 < x < 150):
      count + = 1
```

Теперь более сложная задача: требуется найти средний рост этих детей. Для этого нужно дополнительно в отдельной переменной складывать все нужные значения, а после завершения цикла разделить эту сумму на общее количество значений. Начальное значение переменной **sum**, в которой накапливается сумма, тоже должно быть равно нулю:

```
count = 0
sum = 0
for x in a:
      if 120 < x < 150:
            count + = 1
            sum + = x
print (sum / count)
```

Суммирование элементов списка – это очень распространенная операция, поэтому для суммирования элементов в Python существует встроенная функция sum:

```
a = [1,2,3,4,5]
print (sum(a))
```

С ее помощью можно решить предыдущую задачу более аккуратно: сначала выделить в дополнительный список все нужные элементы, а затем поделить их сумму на количество (длину списка).

Для построения нового списка будем использовать условный оператор:

```
b = [x for x in a if 120 < x < 150]
print (sum(a)/len(a))
```

Условие отбора передало из списка «**a**» в новый список «**b**» только те элементы, которые удовлетворяют условию. А для вывода среднего роста детей в классе остается разделить сумму элементов нового списка на их количество.

КОМПЬЮТЕРНЫЙ ПРАКТИКУМ:

1) *В списке хранятся оценки учеников. Напишите программу, которая выводит номера элементов списка, равных введенной с клавиатуры оценке.*
2) *На физкультуре рост учеников записали в список. Найдите в этом списке ученика с самым высоким и низким ростом.*

11. Тема:

Строки и операции с ними

Изначально компьютеры использовались как вычислительные машины, тогда как сейчас их главной задачей становится обработка текстовой информации. Основной тип данных для работы с текстом в языке Python – это строки (тип **str**) от англ. ***string.***

Строка – это последовательность символов, заключенных в одинарные или двойные кавычки: `'Это строка'` = `"Это строка"`

В отличие от списков (массивов), строки не относятся к структурам данных. При этом строки можно рассматривать как упорядоченную последовательность элементов, с которыми можно работать так же, как и с элементами в списках.

```
>>> s = 'Это строка'
>>> s[0] #возвращает элементы под указанным индексом
'Э'
>>> s[5:] #возвращает элементы с 5-индекса и до последнего
'трока'
```

Однако строки в Python неизменяемы. То есть нельзя заменить какой-то отдельный элемент строки на другой. В этом случае программа выдаст ошибку.
Можно составить из символов существующей строки новую строку, и при этом внести нужные изменения. Приведем полную программу, которая вводит строку с клавиатуры, заменяет в ней все буквы «а» на буквы «б» и выводит полученную строку на экран.

```
s = input('Введите строку: ')
s1 = ''
for c in s:     #перебирает все символы в строке s
    if c == 'а': #если значение переменной совпадает с «а»
        c = 'б'  #то заменяем ее на букву «б»
    s1 = s1 + c
print (s1)
```

Однако этот способ работает очень медленно. В практических задачах, где требуется замена символов, лучше использовать встроенный метод **replace**.

Кроме этого часто требуется строку текста обработать: получить часть этой строки (подстроку), объединить две строки в одну, либо удалить часть строки. Например, для **объединения** (сцепления) строк используется оператор '+'. Эта операция называется конкатенация:

```
s1 = 'Добрый'
s2 = 'день'
s = s1 + ' ' + s2 + '!'
print (s)
>>>
Добрый день!
```

ВСПОМНИ

Длина строки определяется с помощью свойства строки len (англ. length – длина). Для этого в переменную n записывается длина строки s, которая выдает количество знаков вместе с пробелами (целое число):

n = len(s)

Использование срезов для обработки строк

В языке Python часто используются срезы (англ. slicing), которые обозначают определенный участок строки для обработки. Запись среза выглядит так: [X:Y]. X – начало среза, а Y – окончание. Символ с номером Y в срез не входит. По умолчанию первый индекс равен 0, а второй – длине строки.

Например, s[3:8] означает символы строки s с 3-го по 7-й (то есть до 8-го, не включая его).

Значение	Примеры для строки s = 'Понедельник'

Для выделения части строки (подстроки)	`s1 = s[2:8]`
	`#в строку s1 будет записано значение с 3-его элемента по 8-й включительно`
	`>>> недель`

Для удаления части строки	`s1 = s[:3] + s[9:]`
	`#вырезаем с начала 3-элемента, и с 9-ого элемента до конца и складываем их в строке s1`
	`>>> Поник`

Вставить новый фрагмент внутрь строки	`s1 = s[:3] + 'ABC' + s[3:]`
	`#после 3-его элемента вставляем ABC`
	`>>> ПонABCедельник`

Реверс строки (развернуть её наоборот):	`s1 = s[::-1]` `>>> киньледеноП`

Выбрать элементы через заданный шаг	`s1 = s[::2]`
	`#возвращает каждый второй символ`
	`>>> Пндлнк`

Методы строк

В Python есть множество встроенных методов для работы со строками. Рассмотрим наиболее интересные из них.

1 Методы **upper** и **lower** позволяют перевести строку соответственно в верхний и нижний регистр, метод **title** переводит только первые буквы в верхний регистр, а все остальные в нижний:

```
s = 'aAbB cC'
s1 = s.upper() #'AABB CC'
s2 = s.lower() #'aabb cc'
s2 = s.title() #'Aabb Cc'
```

2 Метод **split** позволяет разбить строку по пробелам. В результате получается список из слов. Если пользователь вводит в одной строке ряд слов или чисел, каждое из которых должно в программе обрабатываться отдельно (как в списках), то без split() не обойтись:

```
s = 'Понедельник Вторник Среда Четверг Пятница'
s1 = s.split () #['Понедельник', 'Вторник', 'Среда', 'Чет-
верг', 'Пятница']
```

3 Метод **join**, наоборот, формирует из списка строку. Для этого впереди ставится строка-разделитель, а в скобках передается список:

```
s = ['Понедельник', 'Вторник', 'Среда', 'Четверг', 'Пятница']
s1 = '-'.join (s) #Понедельник-Вторник-Среда-Четверг-Пятница
```

4 Метод **find** работает с подстроками. Он ищет подстроку в строке и возвращает индекс первого элемента найденной подстроки. Если подстрока не найдена, то возвращает -1.

```
s = 'Понедельник Вторник Среда Четверг Пятница'
s1 = s.find ('Среда') #ответ: 20, индекс буквы «С», первого
элемента в подстроке
```

Также этим методом можно найти номер индекса элемента строки в заданном срезе. Для того чтобы указать срез, нужно указать его начало и конец цифрами через запятую. Если вторая цифра не указана, то поиск ведется до конца строки:

```
s2 = s.find ('н', 4) #8, индекс первого «н», в отрезке с
4-ого индекса и до конца
```

```
s1 = s.find ('н', 0, 9) #2, индекс первого «н», в отрезке от начала и 9-ого индекса
```

5 Метод replace заменяет одну подстроку на другую. При этом исходная строка не меняется, а модифицируется новая строка, которая присваивается новой переменной s1:

```
s = 'Понедельник Вторник Среда Четверг Пятница'
s1 = s.replace ('н', 'N') #ПоNедельNик ВторNик Среда Четверг ПятNица
```

Иногда нам нужно разделить строку, чтобы ее часть выводилась с новой строки, тогда мы используем знак \n:

```
print ('Понедельник \n Вторник \n Среда \n Четверг') #все слова выведутся в столбик
```

Если новую строку нужно вывести с отступом, тогда используем знак \t:

```
print ('Понедельник \n\t Вторник \n\t Среда') #все слова выведутся в столбик, каждая новая строка напечатается с отступом
```

Рассмотрим пример обработки строк с использованием изученных выше команд.

Задача 1. С клавиатуры вводится строка, содержащая имя, отчество и фамилию человека, например:

Айтматов Чынгыз Торекулович

Каждые два слова разделены одним пробелом, в начале строки пробелов нет. В результате обработки должна получиться новая строка, содержащая фамилию и инициалы:

Айтматов Ч.Т.

Решение:

1 Введем строку с клавиатуры:

```
s = input('Введите фамилию, имя и отчество: ')
```

2 Разобьем введенную строку на отдельные слова, которые разделены пробелами. Для этого используем метод **split.** В этом списке будут три элемента: **fio[0]** – фамилия, **fio[1]** – имя и **fio[2]** – отчество:

```
fio = s.split()
```

3 Соберем фамилию с инициалами:

```
fioshort = fio[0]+' '+fio[1][0]+'.'+ fio[2][0]+ '.'
```

Полная программа будет выглядеть так:

```
s = input('Введите фамилию, имя и отчество: ')
fio = s.split ()
fioshort = fio[0] + ' ' + fio[1][0] + '.' + fio[2][0] + '.'
print (fioshort)
```

Сравнение и сортировка строк

Мы пользуемся сортировкой по алфавиту для того, чтобы быстрее найти слово. Если бы в словарях слова стояли не по алфавиту, мы бы тратили кучу времени на поиск одного слова. Каков же принцип сортировки строк в Python?

Оказывается, как и числа, буквы имеют свой вес. Буква, стоящая раньше в алфавите – легче, то есть «а» легче, чем «б», поэтому при сортировке она выходит первой. Из этого следует, что строки, как и числа, можно сравнивать.

При сравнении слов сначала сравниваются первые буквы, если они отличаются, то определяется результат сравнения. Дальнейшие буквы уже не сравниваются. Если первые буквы равны, то сравниваются следующие два элемента, и так до конца. Например, слово «паровоз» будет «меньше», чем слово «пароход»: они отличаются в пятой букве и «в» < «х».

Если у вас закончились символы для проверки, то более короткая строка меньше длинной, поэтому «пар» < «парк». Заглавные буквы легче строчных, потому что стоят раньше в алфавите. Поэтому «А» < «а», «П» < «а».

Но откуда компьютер «знает», что такое «алфавитный порядок»? Оказывается, при сравнении строк используются коды символов ASCII и Unicode, где каждый символ имеет свой порядковый номер.

Возьмем пару «ПАР» и «Пар». Первый символ в обоих словах одинаков, а второй отличается – в первом слове буква заглавная, а во втором – такая же, но строчная. Таким образом, получается, что:

«ПАР» < «Пар» < «пар»

А как же с другими символами (цифрами, латинскими буквами)? Цифры стоят в кодовой таблице по порядку, причем раньше, чем латинские буквы; латинские буквы – раньше, чем русские; заглавные буквы (русские и латинские) – раньше, чем соответствующие строчные. Поэтому

«5STEAM» < «STEAM» < «Steam» < «steam» < «ПАР» < «Пар» < «пар»

Например, для того чтобы отсортировать буквы внутри слова, программу можно записать так:

```
s = 'Понедельник'
s1 = ''.join (sorted (s))
print (s1)
>>> Пдееиклнноь
```

Задача 1. Необходимо ввести с клавиатуры несколько слов (например, фамилий) и вывести их на экран в алфавитном порядке.

Для решения этой задачи удобно перезаписать введенные через запятую фамилии в список, а затем отсортировать с помощью метода sorted:

```
s = input('Введите фамилии: ') #Абакиров Муканова Бебинов
Семенова Запруда
s1 = s.split() #формирует список из подстрок ['Абакиров',
'Муканова', 'Бебинов', 'Семенова', 'Запруда']
s2 = ' '.join (sorted (s1))
print (s2)
>>> Абакиров Бебинов Запруда Муканова Семенова
```

ВОПРОСЫ И ЗАДАНИЯ:

1) Напишите программу, которая запрашивает ввести несколько слов с клавиатуры, а затем определяет длину самого короткого слова в строке.

2) Строковый метод isdigit() проверяет, состоит ли строка только из цифр. Напишите программу, которая запрашивает с ввода два целых числа и выводит их сумму. В случае некорректного ввода программа не должна завершаться с ошибкой, а должна продолжать запрашивать числа. Обработчик исключений try-except использовать нельзя.

12. Тема:

Форматирование строк

Форматирование строк в Python означает подстановку в какое-либо место шаблона другого текста. Подстановка происходит, что называется, «на лету». Например, с использованием форматирования можно создавать такие пригласительные билеты через готовый шаблон:

Уважаемый (ая) ***Алина****!*
Приглашаем Вас на ***День открытых дверей****.*
Дата события: ***1 мая***
С уважением, ***Тимур****.*

Одна подстановка запишется так:

```
print ('Уважаемый (ая), {}!'.format ('Алина'))
```

То есть, после текста шаблона вставляются пустые фигурные скобки и далее **.format ()** с указанием в скобах значения, которое нужно вставить. При исполнении программы текст-подстановка вставляется на место фигурных скобок.
При множественных вставках в фигурные скобки вставляются индексы слов, которые идут в кортеже после **.format ().** Таким образом, запись шаблона для нашего приглашения будет выглядеть так:

```
print ('Уважаемый (ая), {0}! \nПриглашаем Вас на {1}.\nДа-
та события: {2} \n С уважением, {3}.'.format ('Алина', 'День
открытых дверей', '1 мая', 'Тимур'))
```

Форматировать текст можно другим способом – без использования .format (). Этот способ считается менее правильным и устаревающим. Однако, если вы встретите в коде оператор % – то, возможно, речь идет именно о форматировании. Запишем наш шаблон приглашения с помощью оператора %:

```
print ('Уважаемый (ая), %s! \nПриглашаем Вас на %s.\nДата
события: %d %s \n С уважением, %s.' % ('Алина', 'День откры-
тых дверей', 1, 'мая', 'Тимур'))
>>>
Уважаемый (ая) Алина!
Приглашаем Вас на День открытых дверей.
Дата события: 1 мая
С уважением, Тимур.
```

Вы наверное заметили, что где-то мы записали %d, где-то %s? Они определяют, что мы используем в качестве подстановки:

%s – подставляет строку;
%d – подставляет целое число;
%f – подставляет дробное число.

Преобразования число-строка и строка-число

В практических задачах часто нужно преобразовать число, записанное в виде цепочки символов, в числовое значение и наоборот. Для этого в языке Python есть стандартные функции:

int – преобразует строку в целое число

```
s = '123'
N = int ( s ) #N = 123
```

float – преобразует строку в вещественное число (дробное)

```
s = '123.456'
X = float ( s ) #X = 123.456
```

str – переводит целое или вещественное число в строку

```
N = 123
s = str ( N ) #s = '123'
X = 123.456
s = str ( X ) #s = '123.456'
```

Если строку не удалось преобразовать в число (например, если в ней содержатся буквы), возникает ошибка и программа завершается.
Для дробных чисел (тип float) можно указать, сколько знаков в дробной части мы хотим вывести, например:

```
X = 34.8589578
print('{:.2f}'.format(X))    #34.86
print('{:.3f}'.format(X))    #34.859
```

Если в больших числах мы хотим использовать запятую в качестве разделителя разрядов, то записываем это так:

```
print('{:,.2f}'.format(10001.23554))    #10,001.24
```

КОМПЬЮТЕРНЫЙ ПРАКТИКУМ:

Напишите шаблон по запросу у пользователя логина и пароля. Если логин или пароль будут введены с ошибкой, то необходимо вывести сообщение о том, что такой логин или пароль не найдены. Если логин и пароль совпадают, то вывести приветствие с указанием имени пользователя.

13. Тема:

Работа с графикой в Python

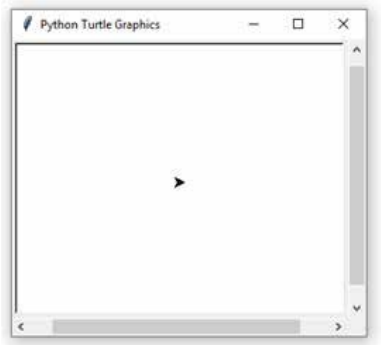
Рисование с помощью модуля Turtle

Для простого рисования в среде программирования Python используется модуль **Turtle** (черепашка), который подключается командой:

```
from turtle import*  #загружает команды для работы с черепашкой
reset () #обнуляет позицию и включает перо
```

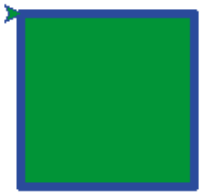
Запустив программу, вы увидите окно для графики с пером (черепашкой) по центру.
Теперь, задавая команды, мы можем строить рисунки на экране. Например, команда **forward** служит для перемещения черепашки вперед, где в скобках указывается количество шагов в пикселях. Для команд поворота **right** и **left** в скобках указывается количество градусов, на которые необходимо повернуть. Попробуем нарисовать квадрат:

```
from turtle import*
reset ()
forward (100) #движение вперед на 100
пикселей
right (90)    #поворот направо на 90˚
forward (100)
right (90)
forward (100)
right (90)
forward (100)
```



Для сокращения кода используем цикл **for**, а также зададим дополнительные параметры для фигуры:

```
from turtle import*
width (5)          #ширина линии - 5 пикселей
color('blue','green') #первый - цвет линии, второй - цвет заливки
begin_fill() #начать следить за черепашкой для заполнения области
for i in range (4):
    forward (100)
    right (90)
end_fill() #заполнить цветом область, начиная с begin_fill()
>>>
```



Используя команду **up()**, вы можете поднять перо и с другого места начать рисовать другой рисунок (или напечатать текст). Для начала рисования нужно опустить перо командой **down().**

Для того чтобы нарисовать круг используется команда **circle(r, n)**, где r – радиус круга, n – часть окружности, которую мы рисуем, в градусах. Если n = 180 градусов, то перо выведет полуокружность, при 360 градусах нарисует полную окружность.

Для того чтобы нарисовать точку, используется команда **dot(r, color),** где r – радиус точки в пикселях, color – цвет, которым будет рисоваться точка.

Задача 1. Нарисуем окружность, по длине которой равномерно распределено заданное количество точек:

```
from turtle import*
def circ(d, r, rBig):  #функция circ с параметрами: кол-во точек,радиус точки, радиус окружности
    for i in range(d):
        circle(rBig, 360 / d)#распределяем кол-во точек по кругу
        dot(r, 'red')
up()
goto(150, 0) #отводим перо на 150 пикселей направо
setheading(90) #поворачиваем перо на 90°
down()         #опускаем перо для начала рисования
circ(15, 10, 150) #передаем значения параметрам
screen.mainloop() #останавливаем выполнение программы
```

Кроме вывода различных фигур, в окне для графики можно рисовать текст. Для этого используется команда **write()** с множеством параметров:

```
write(text, move, align, font = (fontname, fontsize, fontstyle))
```

- в параметр **text** пишется сам текст в кавычках;
- **align** принимает значения «left», «right», «center» и изменяет положение текста относительно черепашки; значения пишутся в кавычках;
- **font** принимает значения fontname, fontsize, fontstyle:
  - в **fontname** пишется название шрифта в кавычках;
  - **fontsize** отвечает за размер шрифта;
  - **fontstyle** отвечает за стиль текста (**normal** – обычный, **bold** – полужирный, **italic** – курсивный, **bold italic** – полужирный текст).

Задача 2. Напишем программу для вывода форматированного текста:

```
from turtle import*
color ('blue')
write('Мы изучаем Python', 'center', font=('Roboto', 24, 'bold italic'))
>>>
```

Команды **clear()** и **reset()** очищают экран от рисунков и перемещают перо в центр.

Работа с Tkinter для создания графических объектов

Модуль Tkinter для работы с графической библиотекой Tk позволяет создавать программы с более продвинутой графикой и анимацией. Так же как и с модулем turtle, модуль tkinter нужно сначала подключить командой:

```
from tkinter import*.
```

Задача 3. Попробуем для начала создать обычную кнопку. Для этого мы используем виджет Button, где в скобках задаем параметры кнопки. С помощью значения text устанавливаем надпись на кнопке. Запишем программу:

```
from tkinter import*
tk = Tk() #создадим форму Tk (объект) с именем tk
btn = Button(tk, text='Пуск')
#создадим кнопку с текстом «Пуск»
btn.pack()  #располагаем кнопку внутри окна
>>>
```

ЭТО ИНТЕРЕСНО!

Виджеты – это базовые блоки для создания графического интерфейса программы, некоторые из которых стандартны во всех языках программирования. Например, это виджеты кнопок, флажки или полоса прокрутки. Они могут лишь отличаться названиями: например, классические флажки (check box) в Tkinter называются check button.

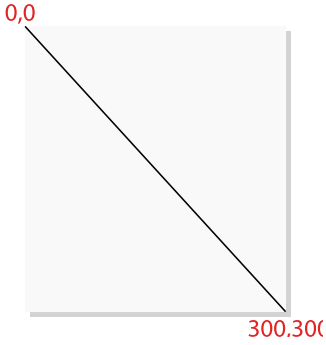
Однако при нажатии этой кнопки ничего не происходит. Для того чтобы нажатие сопровождалось каким-то действием, введем в программу функцию, результат которой выведется через команду **command**:

```
from tkinter import*
def btn_act(): #добавляем функцию, которая выводит на консоли
    print('Игра началась!') #слова «Игра началась!»
tk = Tk()
btn = Button (tk, text='Пуск', command=btn_act) #при нажатии
на кнопку выводится сообщение из функции
btn.pack()
```

Теперь, после нажатия на кнопку, в консоли выведется сообщение:

```
>>> «Игра началась!»
```

Другой виджет – Canvas (*англ. холст*) позволяет рисовать на заданном холсте. Рисование начинается с указания размеров холста, а именно с указания ее ширины (width) и высоты (height).
Для обозначения стартовой точки рисунка на холсте используются координаты X и Y.
Координаты определяют, насколько пиксель (точка) отступает от левого края холста по горизонтали (X) и от верхнего края холста по вертикали (Y).



Для создания линии используется метод **create_line()**, где в скобках указываются четыре числа. Первые два числа – это координаты начала линии, вторые два – координаты конца линии. Запишем программу:

```
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=300, height=300)
canvas.pack()
canvas.create_line(0, 0, 300, 300)
```

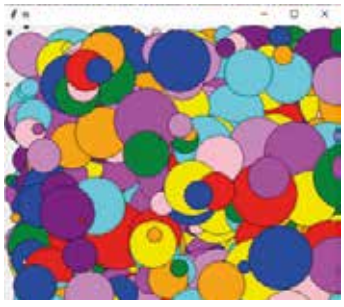
Для создания круга используется метод **create.oval()**:

```
canvas.create_oval(10, 10, 80, 80, outline= 'red', fill=
'green', width=2)
```

В данной записи первые 4 параметра определяют ограничивающие координаты фигуры. Другими словами, это x и y координаты верхней левой и правой нижней точек квадрата, в который помещен круг.

Задача 4. Напишем программу для создания на холсте кругов, диаметры и цвета которых выбираются случайным образом.

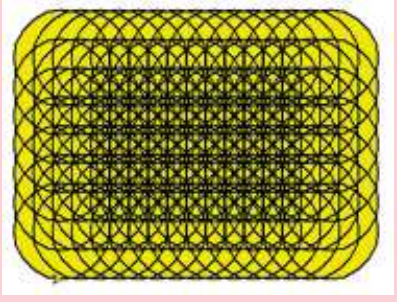
```
from random import* #загружаем функцию randint и choice из
модуля random
from tkinter import*
size = 500 #вводим переменную size
tk = Tk()
diapason = 0 #вводим переменную diapason для ограничения
кол-ва кругов
my_canvas = Canvas (tk, width=size, height=size) #создаем
холст, используя значение переменной size
my_canvas.pack()#располагаем холст внутри окна
while diapason <1000: #цикл повторяется до этого условия
    color = choice(['green', 'red', 'blue', 'orange',
'yellow', 'pink', 'purple', 'violet', 'magenta', 'cyan'])
#создаем список для случайного выбора цветов кругов
    x1 = randint(0,size) #случайный выбор координат x и y
    y1 = randint(0,size)
    d=randint(0,size/5) #произвольный выбор диаметра круга,
но не более size/5
    my_canvas.create_oval(x1,y1,x1+d,y1+d,fill=color)#создаем
круги и заливаем случайным цветом
    tk.update()
    diapason+=1 #шаг цикла, счетчик
```



КОМПЬЮТЕРНЫЙ ПРАКТИКУМ:

1) Нарисуйте с помощью модуля turtle ковер из кругов, где круги имеют один цвет, а фон ковра – другой.

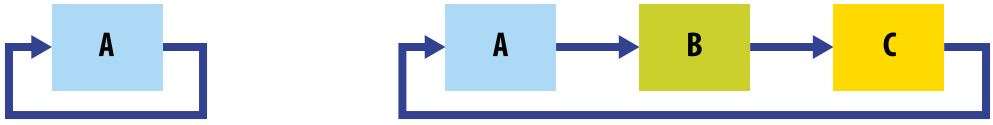
2) Напишите с помощью модуля tkinter программу, которая выводит на холст линии, толщина и цвет которых выбираются случайным образом.



14. Тема:

Рекурсия

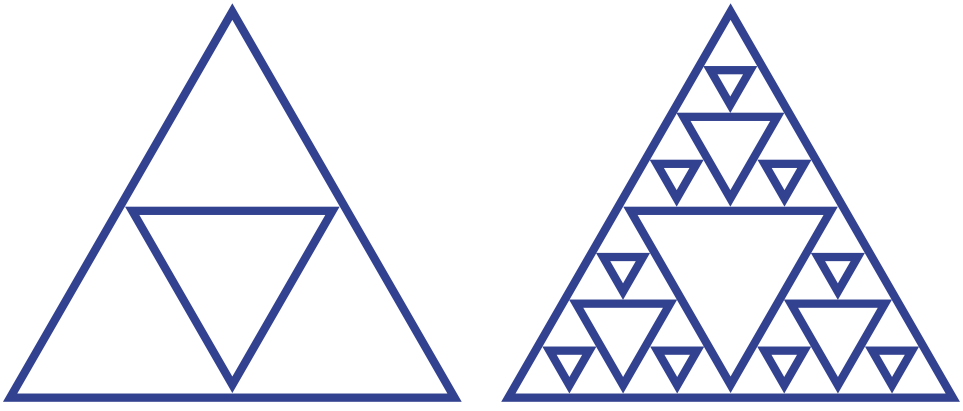
Из курса 8 класса вы помните, что часто одна функция может вызывать другую функцию. Но функция также может вызывать и саму себя. Ситуация, когда программа вызывает сама себя непосредственно (простая рекурсия) или косвенно (через другие функции), например, функция A вызывает функцию B, а функция B – функцию A называется **рекурсией**.



Заметим, что при косвенном обращении все функции в цепочке – рекурсивные.

Популярные примеры рекурсивных объектов – ***фракталы***. Так в математике называют геометрические фигуры, обладающие ***самоподобием***. Это значит, что они составлены из фигур меньшего размера, каждая из которых подобна целой фигуре.

Рис.1. Треугольник Серпинского (один из первых фракталов, предложенных в 1915 году польским математиком В. Серпинским).



Равносторонний треугольник делится на 4 равных треугольника меньшего размера (левый рисунок), затем каждый из полученных треугольников, кроме центрального, снова делится на 4 еще более мелких треугольника и т.д.

Задача 1. Рассмотрим, как работает рекурсия на примере вычисления факториала для n!. Например, чтобы вычислить факториал числа 3, нужно 3*2*1, то есть число n*(n-1)*((n-1)-1).

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n
- 1)
print(factorial(3))

>>>
6     #результат
```

ВСПОМНИ

Факториалом числа n называется произведение всех натуральных чисел от 1 до n:

n!=1*2*3*4*...*n

Например, факториал числа 5 равен 120 (5!=1*2*3*4*5).

Рекурсивные функции решают много задач в программировании, но к сожалению, при их написании часто возникают ошибки. Одна из распространенных ошибок – это бесконечная рекурсия, когда цепочка вызовов функций никогда не завершается и продолжается, пока не закончится свободная память в компьютере.

Две наиболее частые причины для бесконечной рекурсии:

1. Неправильная запись выхода из рекурсии. Например, если мы в программе вычисления факториала забудем поставить проверку `if n == 0`, то factorial(0) вызовет factorial(-1), тот вызовет factorial(-2) и т. д.

2. Неправильная запись параметров в функции. Например, если функция factorial(n) будет вызывать factorial(n), то также получится бесконечная цепочка.

Поэтому при разработке рекурсивной функции в первую очередь нужно оформить условие завершения рекурсии, которое также называется ***базовым условием рекурсии***.

Прямой и обратный ход рекурсии

Действия, выполняемые функцией до входа на следующий уровень рекурсии, называются выполняющимися на прямом ходу рекурсии, а действия, выполняемые по возврату с более глубокого уровня к текущему, – выполняющимися на обратном ходу рекурсии.

Задача 2. Составим процедуру, которая переводит натуральное число в двоичную систему. Стандартный алгоритм перевода числа в двоичную систему можно записать, например, так:

```
def printBin ( n ):
    while n!= 0:
        print (n % 2, end = '')
        n = n // 2
printBin(14)  #для вызова функции введем для примера число 14
>>>
0111  #результат
```

Проблема в том, что двоичное число выводится «задом наперед», то есть первым будет выведен последний разряд.

Есть разные способы решения этой задачи, которые сводятся к тому, чтобы запоминать остатки от деления (например, в символьной строке) и затем, когда результат полностью получен, вывести его на экран.

Пробуем использовать рекурсию. Идея такова: чтобы вывести двоичную запись числа n, нужно сначала вывести двоичную запись числа n // 2, а затем вычислить ее остаток (n%2). Если полученное число – параметр равно нулю, нужно выйти из процедуры. Возьмем для примера число 14:

14 // 2 = 7, 7 % 2 = 1
7 // 2 = 3, 3 % 2 = 1
3 // 2 = 1, 3 % 2 = 1
1 // 2 = 0, выход

Такой алгоритм очень просто программируется. Запишем функцию с названием `print_bin`:

```
def print_bin ( n ):
    if n == 0: return 0
    print_bin ( n // 2 )
    print ( n % 2, end = '')  #записывает остатки в одну
строку
printBin (14)      #для вызова функции введем для примера
число 14

>>>
1110  #результат
```

То же самое можно было сделать и с помощью цикла. Отсюда следует важный вывод: ***рекурсия заменяет цикл.*** При этом программа во многих случаях становится более понятной.

Задача 3. Запишем рекурсивную функцию для возведения числа a в степень b (a**b).

При возведении числа в степень, алгоритм делает следующее:
(a *…(a * (a * (a *(a * 1)))))

Из курса математики мы знаем, что a**0 = 1.

ЗАПОМНИ

Для остановки вывода результатов бесконечной рекурсии нужно нажать на **Ctrl+C**.

Решение задачи:

```
def func_step(a, b):
    if b == 0:
        return 1
    else:
        return a * func_step(a, b-1)
print(func_step(2, 4))    #например, 2 в 4 степени
>>>
16    #результат
```

ЭТО ИНТЕРЕСНО!

Примеры рекурсии в жизни человека

- Вы положили деньги в банк под проценты и начисленные проценты остаются в банке на вашем счете и на них начисляют проценты. Процесс продолжается пока вы не заберете весь вклад.
- Повторяющееся отражение в зеркалах, расположенных напротив друг друга – это бесконечная рекурсия.

КОМПЬЮТЕРНЫЙ ПРАКТИКУМ:

1) Дано натуральное число n. Выведите все числа от 1 до n.
2) Запишите рекурсивную функцию, которая для заданного числа n вычисляет сумму всех чисел от 1 до n.

15. Тема:

Алгоритмы обработки массивов

В 7 и 8 классах вы уже начали изучать массивы, представленные в виде списков, кортежей и словарей, исполнять некоторые операции с ними, вводить и выводить данные из массива.

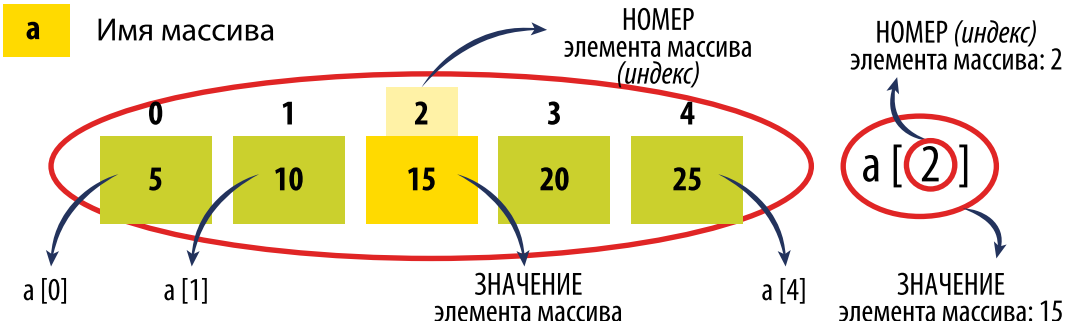
В этой теме мы более подробно рассмотрим основные стандартные алгоритмы для обработки данных в массивах:

- поиска
- модификации
- сортировки

СМОТРИ ТАКЖЕ
Тема 3.6 *8 класс*
Массивы

ВСПОМНИ

Массив – это группа переменных, расположенных в памяти рядом (в соседних ячейках) и имеющих общее имя. Каждая ячейка имеет уникальный номер – индекс.



Поиск в массиве

Рассмотрим алгоритм поиска на примере задачи, в которой **a** – имя массива, **n** – количество элементов в массиве (целое число), а переменная i будет обозначать индекс элемента списка.

Задача 1. Необходимо найти в массиве элемент, равный значению переменной **x** или сообщить, что его нет. Алгоритм решения – это просмотр всех элементов массива с первого до последнего. Как только найден элемент, равный **x**, нужно выйти из цикла и вывести результат.

Для этого мы используем цикл с переменной **nx**, перебираем все элементы массива и досрочно завершаем цикл, если найдено требуемое значение.

```
a = [1, 5, 4, 31, 10] #Задан массив с вариантами значений
x = int(input('Введите X: '))  #Ввод искомого значения x
nx = 0    #Переменная nx сохраняет номер найденного элемента
for item in a:          #для элементов массива a
    if item == x:       #если элемент будет равен x
        nx = item       #то оно запишется в переменную nx
        break           #завершает цикл
if nx > = 1:            #переменная nx – изменилась
    print ('Нашли!')  #выдает что искомое значение найдено
else:
    print ('Не нашли')
```

Для выхода из цикла используется оператор **break**, номер найденного элемента сохраняется в переменной «**nx**». Если ее значение осталось равным 0 (не изменилось в ходе выполнения цикла), то в массиве нет элемента, равного **x**.

Эту задачу можно решить еще одним способом: использовать функцию **index** для массива, которая возвращает индекс первого найденного элемента равного **x**:

```
a = [16, 29, -5, -11, 23, 14, -7, 23, 18] #пример массива
print (a.index (23))
>>>
4    #результат
```

Если элемента в массиве нет, программа вернет ошибку и прекратит работу.

Задача 2. Найдем в массиве максимальный элемент, который должен записаться в переменную «**m**». Для этого надо просмотреть все элементы массива один за другим. Если очередной элемент массива больше, чем максимальный из предыдущих (находящийся в переменной «**m**»), запомним новое значение максимального элемента в «**m**».

Поскольку содержимое массива неизвестно, можно записать в «**m**» ноль или отрицательное число. Если это будет **m = a[0]**, то цикл перебора начнется со второго элемента, то есть с **a[1]**:

```
a = range (1, 20)
m = a[0]
for i in range (1, 20):
    if a[i] > m:
        m = a[i]
print (m)

Вот еще один вариант:
a = range (1, 20)
m = a[0]
for x in a:
    if x > m:
        m = x
print (m)
```

Он отличается тем, что не использует переменную-индекс, но зато дважды просматривается элемент **a[0]** (второй раз – в цикле, где выполняется перебор всех элементов).

Поскольку операции поиска максимального и минимального элементов нужны очень часто, в Python есть соответствующие встроенные функции **max()** и **min()**:

```
a = range (1,20)
m = max (a)
print (m)
```

Модификация массива

Для работы с массивами и изменения в них данных в Python есть множество различных функций, например:

ФУНКЦИЯ	ЗНАЧЕНИЕ	ПРИМЕР

print (a)	Выводит список a на экран	`a = [16, 'b', 34, 'c']#базовый список для всех примеров`
		`print (a)`
		`>>> [16, 'b', 34, 'c']`

append ()	Добавляет один элемент в конец списка	`a.append (18)`
		`print (a)`
		`>>> [16, 'b', 34, 'c', 18]`

clear ()	Удаляет все элементы списка	`a.clear ()`
		`print (a)`
		`>>> []`

ФУНКЦИЯ	ЗНАЧЕНИЕ	ПРИМЕР
count ()	Возвращает количество элементов с заданным значением	`print (a.count (16))#считает сколько элементов со значением 16 есть в списке` `>>> 1`

extend ()	Добавляет другой список в конец базового списка	`B = [18, 'h'] #второй список`
		`a.extend (b)`
		`print (a)`
		`>>> [16, 'b', 34, 'c', 18, 'h']`

index ()	Возвращает номер индекса первого найденного похожего элемента	`print (a.index (34))` `>>> 2`

insert ()	Вставляет элемент по индексу	`a.insert (1, 22) #поскольку отсчет индекса идет с 0, то в базовом списке под индексом 1 стоит элемент 'b'; вместо него должна вставиться цифра 22, остальные сдвигаются, вправо.`
		`print (a)`
		`>>> [16, 22, 'b', 34, 'c']`

pop ()	Удаляет элемент под заданным индексом	`a.pop (0) #удалить значение под индексом 0`
		`print (a)`
		`>>> ['b', 34, 'c']`
		
		`a.pop ()`
		`print (a)`
		`>>> [16,'b', 34] #если значение не задаются, то удаляется последний элемент`

remove ()	Удаляет первый элемент с заданным значением. Если элемент не найден – то выдается ValueError	`a.remove (34)`
		`print (a)`
		`>>> [16, 'b', 'c']`

reverse ()	Переставляет элементы списка в обратном порядке	`a.reverse ()`
		`print (a)`
		`>>> ['c', 34, 'b', 16]`

sort ()	Сортирует список (только для списков с одним типом элементов)	`a = [16, 8, 34, 3] #в списке только числа(int)`
		`a.sort ()`
		`print (a)`
		`>>> [3, 8, 16, 34] #сортирует по возрастанию`
		
		`a = ['m', 'b', 'o', 'c'] #в списке только символы (str)`
		`a.sort ()`
		`print (a)`
		`#сортирует по алфавиту`
		`>>> ['b', 'c', 'm', 'o']`

Рассмотрим более подробно некоторые из них: реверс массива, сдвиг элементов массива и отбор нужных элементов.

Реверс массива

Реверс массива *(reverse)* – это перестановка его элементов в обратном порядке: первый элемент становится последним, а последний – первым.

Индекс элементов массива в Python начинаются с 0. Поэтому, если общее количество элементов равно N, то противоположный для i элемент находится по формуле: **N - i - 1**. Например (см.таблицу ниже), чтобы найти индекс элемента, который заменит элемент под индексом 2, вычисляем по формуле:

N i
9 – 2 – 1 = 6

Элемент под индексом 2 должен поменяться местами с элементом под индексом 6.

Элементы массива	16	29	-5	-11	23	14	-7	23	18
Индексы	0	1	2	3	4	5	6	7 или (n -2)	8 или (n -1)

Чтобы найти вторую пару элемента, мы просматриваем индексы только первой половины массива. Это значит, что цикл нужно остановить на середине массива:

```
a = [16, 29, -5, -11, 23, 14, -7, 23, 18]
n = len(a)     #вычисляем количество элементов в массиве
for i in range(n//2):#только индексы первой половины массива
    a[i], a[n-i-1] = a[n-i-1],
a[i]  #переставляем элементы
print(a)
>>>
[18, 23, -7, 14, 23, -11, -5,
29, 16]
```

Операция реверс массива может быть выполнена и с помощью стандартного метода **reverse()**:

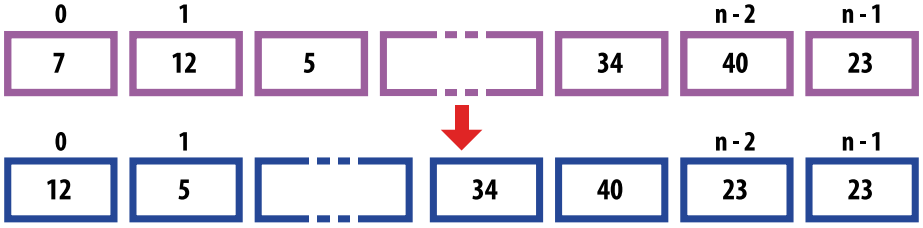
```
a.reverse ()
```

ЗАПОМНИ

Значения индексов в Python могут быть отрицательными. В этом случае нумерация будет идти с конца. Например, индекс самого последнего элемента массива равен -1, предпоследнего -2 и т.д.

Сдвиг элементов массива

При удалении и вставке элементов необходимо выполнять сдвиг части или всех элементов массива в ту или другую сторону. Массив часто рисуют в виде таблицы, где первый элемент расположен слева. Поэтому сдвиг влево – это перемещение всех элементов на одну ячейку, при котором **a[1]** переходит на место **a[0], a[2]** – на место **a[1]** и т.д.



Последний элемент остается на своем месте, то есть дублируется, поскольку новое значение для него взять неоткуда – массив кончился.

Запишем алгоритм:

```
a = [16, 29, -5, -11, 23, 14, -7, 23, 18]
n = len(a)
for i in range (n-1):
    a[i] = a[i+1]
print(a)
>>>
[29, -5, -11, 23, 14, -7, 23, 18, 18]
```

Обратите внимание, что цикл заканчивается при a = [n-1], чтобы не было выхода за границы массива, то есть обращения к несуществующему элементу a[n].

Как видим, первый элемент пропал, а последний – повторился 2 раза. Можно старое значение первого элемента записать на место последнего. Такой сдвиг называется циклическим. Для этого предварительно (до начала цикла) первый элемент нужно запомнить во вспомогательной переменной (c), а после завершения цикла записать его в последнюю ячейку массива:

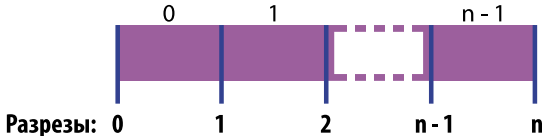
```
a = [16, 29, -5, -11, 23, 14, -7, 23, 18]
n = len(a)
c = a[0]
for i in range(n-1):
    a[i] = a[i+1]
a[n-1] = c
print(a)
>>>
[29, -5, -11, 23, 14, -7, 23, 18, 16]
```

Можно выполнить сдвиг, используя встроенные возможности Python:

```
a = a[1:n] + [a[0]]
```

Здесь первый элемент a[0] ставится в конец к обрезанному массиву a[1:n], который теперь начинается не с нуля, а с единицы.

На рисунке справа мы видим, что срез a[0:2] включает все элементы между 0 и 2, то есть элементы a[0] и a[1].



Отбор нужных элементов

Требуется отобрать все элементы массива **a**, удовлетворяющие некоторому условию, в новый массив **b**. Для этого перебираем все элементы исходного массива и, если очередной элемент нам подходит, добавляем его в новый массив, используя второй счетчик. Н-р, соберем во второй список четные числа:

```
a = [16, 29, -5, -11, 23, 14, -7, 23, 18]
b = []
for x in a:
    if x % 2 == 0:
        b.append(x)
print(b)
>>>
[16, 14, 18]
```

Второй вариант решения – использование генератора с условием

```
a = [16, 29, -5, -11, 23, 14, -7, 23, 18]
b = [x for x in a if x % 2 == 0]
print(b)
```

Здесь в новый список b отбираются только элементы, делящиеся на 2.

КОМПЬЮТЕРНЫЙ ПРАКТИКУМ:

*1) Заполните массив случайными числами в интервале (0,20). Введите число **x** и найдите все значения, равные **x**.*
2) Дан одномерный массив числовых значений, насчитывающий n элементов. Выполните перемещение элементов массива по кругу вправо, т.е. a(1) –> a(2); a(2) –> a(3); ... a(n) –> a(1).

16. Тема:

Сортировка списков

Часто для того чтобы облегчить поиск нужной информации, мы пользуемся сортировкой. Например, сортировка слов по алфавиту облегчает поиск слова в словаре.

В программировании **сортировка** – это перестановка элементов массива в заданном порядке.

Порядок сортировки может быть любым: для чисел обычно рассматривают сортировку по возрастанию (или убыванию) значений. Например, при сортировке по возрастанию из одномерного массива [3 1 0 5 2 7] получается массив [0 1 2 3 5 7]. Символьные данные обычно сортируются в алфавитном порядке.

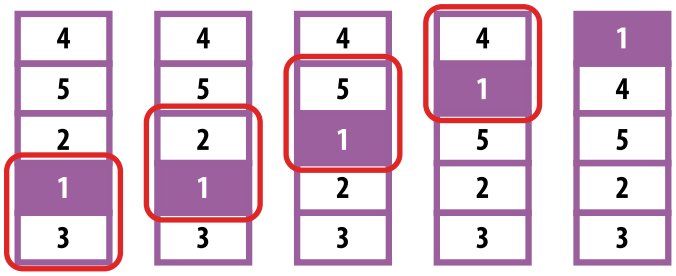
Было придумано множество способов сортировки. В целом их можно разделить на две группы: 1) простые, но медленно работающие (на больших массивах) и 2) сложные, но быстрые.

Рассмотрим один из наиболее наглядных методов сортировки – «метод пузырька».

Метод пузырька (сортировка обменами)

Название этого метода произошло от известного физического явления – пузырек воздуха в воде поднимается вверх. Для сортировки массива таким способом необходимо справа налево пройтись по массиву, попарно сравнивая соседние элементы, и в том случае, когда левый больше правого, менять их местами. Так самые «тяжелые» элементы падают на дно, а самые «легкие» (минимальные) элементы поднимаются наверх (к началу массива) подобно пузырькам воздуха.

Далее так же рассматриваем следующую пару элементов от конца и т.д. *(см. рисунок).*



Когда мы обработали пару (a[0], a[1]), минимальный элемент стоит на месте a[0]. Это значит, что на следующих этапах его можно не рассматривать. Первый цикл, устанавливающий на свое место первый (минимальный) элемент, можно на псевдокоде записать так:

```
для i от n-2 до 0 шаг -1
      if (a[j] > a[j+1])
            поменять местами a[j] и a[j+1]
```

Переменная i хранит индекс ячейки, в которую записывается минимальный элемент. Сначала это будет первая ячейка. Переменная j используется для обращения к текущим сравниваемым ячейкам массива.

За один проход такой цикл ставит на место один элемент. Чтобы «подтянуть» второй элемент, нужно написать еще один почти такой же цикл, который будет отличаться только конечным значением i в заголовке цикла. Так как верхний элемент уже стоит на месте, его не нужно трогать:

```
for i in range(n-1): #количество переходов
    for j in range(n-i-1):
        if a[j] > a[j+1]:
            a[j], a[j+1] = a[j+1], a[j]
```

Таких циклов нужно сделать **n-1** – на 1 меньше, чем количество элементов массива. Почему не n? Дело в том, что если n-1 элементов поставлены на свои места, то оставшийся автоматически встает на свое место – другого места нет.

Запишем полную программу сортировки методом пузырька:

```
from random import randint
n = 10
a = [randint (1,99) for n in range (n)]
print (a)
for i in range (n-1): #кол-во проходов по списку
    for j in range (n-i-1): #кол-во сравнений уменьшается на величину i
        if a [j] > a [j+1]: #вложенный цикл сравнивает эл-т j с j+1
            a[j], a [j+1] = a[j+1], a[j] #при необходимости
меняет местами
print(a)
>>>
[5, 84, 90, 37, 30, 32, 29, 62, 17, 99]
[5, 17, 29, 30, 32, 37, 62, 84, 90, 99]
```

Метод выбора

Еще один популярный и простой метод сортировки – метод выбора. Сначала просматривается весь массив, находится минимальный элемент и переставляется в самое начало массива. Во втором проходе, среди всех оставшихся, то есть со второго по последний элемент, снова находится наименьший элемент и переставляется местом со 2-м. Далее среди элементов, начиная с 3-го, также находится наименьший и меняется с 3-м и так далее до (n-1)-го элемента.

Запишем алгоритм для сортировки списка методом выбора, где переменная i хранит индекс ячейки, в которую записывается минимальный элемент, а переменная j – просматриваемый элемент:

```
for i in range(n-1):
    n_min = i
    for j in range(i+1,n):
        if a[j] < a[n_min]:
            n_min = j
    if i != n_min:
        a[i], a[n_min] = a[n_min], a[i]
```

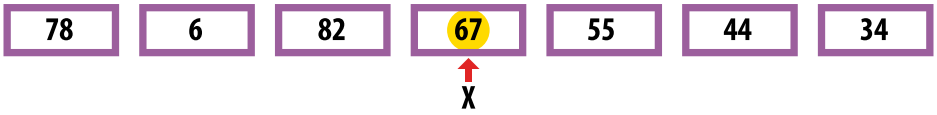
Здесь перестановка происходит только тогда, когда найденный минимальный элемент стоит не на своем месте, то есть **i != n_min**. Поскольку поиск номера минимального элемента выполняется в цикле, этот алгоритм сортировки также представляет собой вложенный цикл.

«Быстрая сортировка» или quicksort

Пузырьковая сортировка и метод выбора работают медленно для больших массивов данных (более 10000 элементов). Поэтому для сортировки больших массивов часто используется рекурсивный алгоритм «быстрой сортировки» (англ. quicksort).

Идея такой сортировки состоит в том, что сначала массив разбивается где-то посередине, затем из элементов левой части выбираются те, которые больше или равны значению «среднего элемента» и переставляются в правую часть. Далее левая и правая части рассматриваются как отдельные массивы и снова делятся на два.

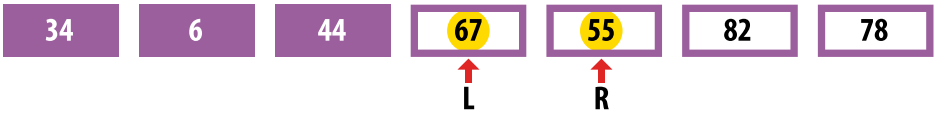
Чтобы понять сущность метода, рассмотрим пример. Пусть задан массив:



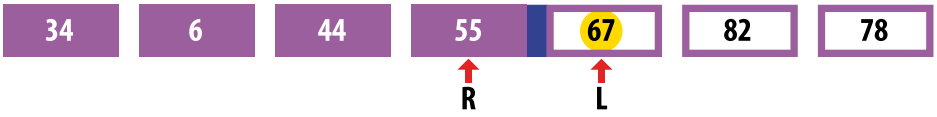
Выберем в качестве **X** средний элемент массива, то есть 67. Найдем первый слева элемент массива, который больше или равен **X** и должен стоять во второй части. Это число 78. Обозначим индекс этого элемента через **L**. Теперь находим самый правый элемент, который меньше **X** и должен стоять в первой части. Это число 34. Обозначим его индекс через **R**.

Теперь поменяем местами два этих элемента. Сдвигая переменную **L** вправо, а **R** – влево, находим следующую пару, которую надо переставить. Это числа 82 и 44.

Следующая пара элементов для перестановки – числа 67 и 55.



После этой перестановки дальнейший поиск приводит к тому, что переменная **L** становится больше **R**, то есть массив разбит на две части. В результате все элементы массива, расположенные левее **A[L]**, меньше или равны **X**, а все правее **A[R]** – больше или равны **X**.



Таким образом, сортировка исходного массива свелась к двум сортировкам частей массива, то есть к двум задачам того же типа, но меньшего размера. Теперь нужно применить тот же алгоритм к двум полученным частям массива: первая часть – с 1-го до **R**-го элемента, вторая часть – с **L**-го до последнего элемента.

Как мы уже говорили, скорость сортировки важна при работе с большими массивами. Ниже в таблице сравнивается время сортировки (в секундах)

массивов разного размера, заполненных случайными значениями, с использованием трех изученных алгоритмов.

N	Метод пузырька	Метод выбора	Быстрая сортировка
1000	0,09 с	0,05 с	0,002 с
5000	2,4 с	1,2 с	0,014 с
15000	22 с	11 с	0,046 с

Как видно из таблицы, при обработке массива, к примеру, с 15 тысячами элементов, быстрая сортировка работает в 500 раз быстрее, чем при методе пузырька.

Из предыдущей темы мы также знаем, что в Python есть встроенная функция для сортировки массивов **sorted**, которая использует гибридный алгоритм ***timesort*** и легко справляется с обработкой большинства известных данных. Необходимо также помнить, что при записи:

```
a.sort ()   #сортируется сам список a.
b = sorted (a) #в массив b переносится отсортированный в порядке возрастания массив a.
```

По умолчанию сортировка выполняется по возрастанию или точнее «неубыванию», когда каждый следующий элемент больше или равен предыдущему. Для того чтобы отсортировать массив по убыванию (невозрастанию – каждый следующий элемент меньше или равен предыдущему), нужно в функции сортировки указать reverse = True:

```
b = sorted (a, reverse = True) #или
a.sort (reverse = True)
```

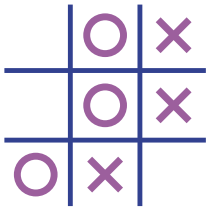
КОМПЬЮТЕРНЫЙ ПРАКТИКУМ:

1) *Дан одномерный массив числовых значений, насчитывающий n элементов. Из элементов исходного массива построить два новых. В первый должны входить только числа, которые делятся на 3, а во второй числа, которые делятся на 5.*
2) *Продолжите программу из первого вопроса и допишите алгоритм, который сортирует числа, делящиеся на 3 по возрастанию, а все числа делящиеся на 5 – по убыванию.*

17. Тема:

Матрицы

Матрица – это двумерный массив, имеющий табличную структуру. Описываются массивы так же, как одномерные. Разница состоит в том, что у элемента двумерного массива две координаты (два индекса) – номер строки и номер столбца, в которых находится элемент.
Например, при составлении программы для игры в крестики-нолики можно пустым клеткам присвоить код «–1», клетке. С ноликом – код 0, а клетке с крестиком – код 1:



	0	1	2
0	-1	0	1
1	-1	0	1
2	0	1	-1

В Python для работы с таблицами используют списки. Двумерная таблица хранится как список, каждый элемент которого тоже представляет собой список («список списков»). Например, таблицу, показанную на рисунке, можно записать так:

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

Данный оператор можно записать в одну строку:

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

Но поскольку человек воспринимает матрицу как таблицу, лучше и на экран выводить ее в виде таблицы. Для этого матрицу можно записать так:

```
a = [[-1, 0, 1], [-1, 0, 1], [0, 1, -1]]
for i in range ( len(a) ):
    for j in range ( len(a[i]) ):
        print ( '{:4d}'.
format(a[i][j]), end = '' )
    print ()
>>>
```

Формат вывода `{:4d}` – раздвигает поля между столбцами: в нашем примере перед каждым элементом ставится 4 пробела.

```
-1    0    1
-1    0    1
 0    1   -1
```

Здесь **i** – индекс подсписка (определяет число строк), а **j** – индекс элемента внутри подсписка (определяет число столбцов); **len(a)** – это число подсписков в большом списке (их здесь 3), **len(a[i])** – число элементов подсписка, которое совпадает с числом столбцов.
a [i] [j] – это элемент в подсписке i, под j-индексом:

`a[0][0]==-1, a[0][1]==0, a[0][2]==1, a[1][0]==-1,` и т.д.

Этот же пример можно записать так:

```
for row in a:           #в строке a
    for elem in row:    #для элемента в строке
        print(elem, end=' ') #вывести элементы
    print()
```

Задача 1. Заполним матрицу случайными цифрами. Количество строк и столбцов введем с клавиатуры.

Каждому элементу матрицы можно присвоить любое значение. Поскольку индексов два, для заполнения матрицы нужно использовать вложенный цикл. Далее будем считать, что существует матрица **a**, состоящая из **n** строк и **m** столбцов, а **i** и **j** – целочисленные переменные, обозначающие индексы строки и столбца. В этом примере матрица заполняется случайными числами и выводится на экран:

```
import random
n = int(input ('Введите количество строк: '))
m = int(input ('Введите количество столбцов: '))
a = []
for i in range (n):
    a.append([])
    for j in range (m):
        a[i].append (random.randint (10,40))#каждому э-ту
присваивается случайное число от 10 до 40
for i in a:     #каждый подсписок выведется с новой строки,
    print (i)   #но в квадратных скобках
>>>
[33, 16, 31, 33]     #случайные числа при n=2, m=4
[39, 35, 11, 15]
```

Эту же задачу запишем короче, и так, чтобы в результате не было скобок:

```
import random
n = int(input ('Введите количество строк: '))
m = int(input ('Введите количество столбцов: '))
a = [[random.randint(10, 40) for i in range(m)] for j in
range(n)]
print(' '.join([str(elem) for elem in row])) #все элементы
объединяются и перечисляются через пробел
```

Обработка двумерного массива

Такой же двойной цикл нужно использовать для перебора всех элементов матрицы. Первый цикл перебирает номер строки, второй цикл бежит по элементам внутри строки. Вот как вычисляется сумма (s) всех элементов:

```
a = [[1, 2, 3, 4], [5, 6], [7, 8, 9]]
s = 0
for i in range(len(a)):
    for j in range(len(a[i])):
        s + = a[i][j]
print(s) #результат 45
```

Для этой записи можно использовать встроенную функцию **sum**:

```
a = [[1, 2, 3, 4], [5, 6], [7, 8, 9]]
s = 0
for row in a:
    s + = sum(row)
print (s)
```

Обработку некоторых элементов матрицы рассмотрим на примере задачи:

Задача 2. Пусть дан квадратный массив из n строк и n столбцов. Необходимо заполнить диагональ единичками, область слева от нее заполнить двойками, а область справа – нулями.

```
1 0 0 0
2 1 0 0
2 2 1 0
2 2 2 1
```

Главная диагональ – это элементы a[0, 0], a[1, 1], …, a[n-1, n-1], то есть номер строки равен номеру столбца. Для заполнения главной диагонали единицами нужен один цикл:

```
for i in range(n):      #работаем с a[i][i]
    a[i][i] = 1         #заполняем их единицами
```

Элементы справа от диагонали заполним значением 0, для чего нам понадобится в каждой из строк с номером i присвоить значение элементам a[i][j] для j=i+1, ..., n-1. Здесь нам понадобятся вложенные циклы:

```
for i in range(n):
    for j in range(i + 1, n):
        a[i][j] = 0
```

Аналогично присваиваем значение 2 элементам a[i][j] для j=0, ..., i-1:

```
for i in range(n):
    for j in range(0, i):
        a[i][j] = 2
```

Если внешние циклы объединить в один, то можно получить одно такое решение:

```
n = 4
a = [[0] * n for i in range(n)]
for i in range(n):
    for j in range(0, i):
        a[i][j] = 2
    a[i][i] = 1
    for j in range(i + 1, n):
        a[i][j] = 0
for row in a:
    print(' '.join([str(elem) for elem in row]))
```

КОМПЬЮТЕРНЫЙ ПРАКТИКУМ:

*1) Заполните прямоугольную матрицу **a**, имеющую **n** строк и **m** столбцов элементами из случайных чисел. Найдите среднее арифметическое элементов массива.*
2) Найдите наибольшее значение среди средних значений для каждой строки матрицы.

Приложение №1

Элемент электронной таблицы	Способ выделения
Ячейка	Установить курсор в ячейку и нажать ЛКМ, при этом изменится цвет имени столбца и номера строки, расположенные на линейках (ячейка становится активной).
Диапазон смежных ячеек	**1 способ:** установить курсор в первую угловую ячейку диапазона, удерживая нажатой клавишу Shift, переместить курсор в противоположный угол диапазона, щелкнуть ЛКМ. **2 способ:** установить курсор в первую угловую ячейку, удерживая ЛКМ, провести по диагонали в противоположный угол диапазона.
Диапазон несмежных ячеек	Удерживая клавишу Ctrl, выделить отдельные диапазоны, используя способы выделения смежных ячеек.
Строка	Щелкнуть по номеру строки на левой линейке.
Столбец	Щелкнуть по имени столбца на верхней линейке.
Лист	Щелкнуть по прямоугольнику, расположенному между заголовком столбца А и заголовком строки 1.

Приложение №2

Арифметические операторы		
Оператор	**Значение**	**Примеры**
+	**Сложение** - суммирует значения слева и справа от оператора	*10 + 5 в результате будет 15* *27 + -3 в результате будет 24* *9.4 + 7 в результате будет 16.4*
-	**Вычитание** - вычитает правый операнд из левого	*15 - 5 в результате будет 10* *20 - -3 в результате будет 23* *13.4 - 7 в результате будет 6.4*
*	**Умножение** - перемножает операнды	*5 * 5 в результате будет 25* *7 * 3.2 в результате будет 22.4* *-3 * 12 в результате будет -36*
/	**Деление** - делит левый операнд на правый	*15 / 5 в результате будет 3* *5 / 2 в результате будет 2.5*
%	**Деление по модулю** - делит левый операнд на правый и возвращает остаток.	*6 % 2 в результате будет 0* *7 % 2 в результате будет 1* *13.2 % 5 в результате 3.19999999*
**	**Возведение в степень** - возводит левый операнд в степень правого	*5 ** 2 в результате будет 25* *2 ** 3 в результате будет 8* *-3 ** 2 в результате будет -9*
//	**Целочисленное деление** – деление, в котором возвращается только целая часть результата. Часть после запятой отбрасывается.	*12 // 5 в результате будет 2* *4 // 3 в результате будет 1* *25 // 6 в результате будет 4*

Операторы сравнения

==	Проверяет, равны ли оба операнда. Если да, то условие становится истинным.	*5 == 5 в результате будет True* *True == False в результате будет False* *"hello" == "hello" в результате будет True*
!=	Проверяет, равны ли оба операнда. Если нет, то условие становится истинным.	*12 != 5 в результате будет True* *False != False в результате будет False* *"hi" != "Hi" в результате будет True*
<>	Проверяет, равны ли оба операнда. Если нет, то условие становится истинным.	*12 <> 5 в результате будет True.* *Похоже на оператор !=*
>	Проверяет, больше ли значение левого операнда чем значение правого. Если да, то условие становится истинным.	*5 > 2 в результате будет True.* *True > False в результате будет True.* *"A" > "B" в результате будет False.*
<	Проверяет, меньше ли значение левого операнда чем значение правого. Если да, то условие становится истинным.	*3 < 5 в результате будет True.* *True < False в результате будет False.* *"A" < "B" в результате будет True.*
>=	Проверяет, больше или равно значение левого операнда чем значение правого. Если да, то условие становится истинным.	*1 >= 1 в результате будет True.* *23 >= 3.2 в результате будет True.* *"C" >= "D" в результате будет False.*
<=	Проверяет, меньше или равно значение левого операнда чем значение правого. Если да, то условие становится истинным.	*4 <= 5 в результате будет True.* *0 <= 0.0 в результате будет True.* *-0.001 <= -36 в результате будет False.*

Операторы сравнения

=	Присваивает значение правого операнда левому.	*b = 23 присвоит переменной b значение 23*
+=	Прибавит значение правого операнда к левому и присвоит эту сумму левому операнду	*b = 5* *a = 2* *b += a равносильно: b = b + a. b=7*
-=	Отнимает значение правого операнда от левого и присваивает результат левому операнду.	*b = 5* *a = 2* *b -= a равносильно: b = b - a. b=3*
*=	Умножает правый операнд с левым и присваивает результат левому операнду.	*b = 5* *a = 2* *b *= a равносильно: b = b * a. b=10*
/=	Делит левый операнд на правый и присваивает результат левому операнду.	*b = 10* *a = 2* *b /= a равносильно: b = b / a. b=5*
%=	Делит по модулю операнды и присваивает результат левому.	*b = 5* *a = 2* *b %= a равносильно: b = b % a. b=1*
**=	Возводит в левый операнд в степень правого и присваивает результат левому операнду.	*b = 3* *a = 2* *b **= a равносильно: b = b ** a. b=9*
//=	Производит целочисленное деление левого операнда на правый и присваивает результат левому операнду.	*b = 11* *a = 2* *b //= a равносильно: b = b // a. b=5*

Арифметические операторы

Оператор	Значение	Примеры
and	Логический оператор "И". Условие будет истинным, если оба операнда истина.	*True and True равно True.* *True and False равно False.* *False and True равно False.* *False and False равно False.*
or	Логический оператор "ИЛИ". Если хотя бы один из операндов истинный, то и все выражение будет истинным.	*True or True равно True.* *True or False равно True.* *False or True равно True.* *False or False равно False.*
not	Логический оператор "НЕ". Изменяет логическое значение операнда на противоположное.	*not True равно False.* *not False равно True.*

Операторы членства

Операторы членства проверяют наличие элемента в составных типах данных, таких как строки, списки, кортежи или словари:

Оператор	Значение	Примеры
in	Возвращает истину, если элемент присутствует в последовательности, иначе возвращает ложь.	*"cad" in "cadillac" вернет True.* *1 in [2,3,1,6] вернет True.* *"hi" in {"hi":2,"bye":1} вернет True.* *2 in {"hi":2,"bye":1} вернет False* *(в словарях проверяется наличие в ключах, а не в значениях).*
not in	Возвращает истину, если элемента нет в последовательности.	*Результаты противоположны результатам оператора in.*

Операторы тождественности

Операторы тождественности сравнивают размещение двух объектов в памяти компьютера.

Оператор	Значение	Примеры
is	Возвращает истину, если оба операнда указывают на один объект.	*x is y вернет истину, если id(x) будет равно id(y).*
is not	Возвращает ложь, если оба операнда указывают на один объект.	*x is not y, вернет истину, если id(x) не равно id(y).*