Оператор присваивания =

Знак равенства ('=') используется для присваивания переменной какого-либо значения. После этого действия в интерактивном режиме ничего не выводится:

```
>>> width = 20
>>> height = 5*9
>>> width * height
900
```

Значение может быть присвоено нескольким переменным одновременно:

```
x = y = z = 0  # Нулевые x, y и z
```

Переменные должны быть *определены* (defined) (должны иметь присвоенное значение) перед использованием, иначе будет сгенерирована ошибка:

```
>>> # попытка получить доступ к неопределённой переменной...
>>> n
Traceback (most recent call last):
File "<stdin>", line 1, in <module>NameError: name 'n' is not defined
```

Ввод данных: функция input()

Функция **input()** считывает строку с клавиатуры и возвращает значение считанной строки, которое необходимо присвоить переменным:

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

Функция input возвращает текстовую строку, а переменные должны быть целочисленные.

Поэтому сразу же после считывания выполним преобразование типов при помощи фунцкии **int**, и запишем новые значения в переменные a и b.

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

Можно объединить считывание строк и преобразование типов, если вызывать функцию **int** для того значения, которое вернет функция input:

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

Ввод данных с «подсказкой»:

input("строка подсказка")

```
>>> a=int( input("введте значение a ") )
введте значение a 55
>>> print(a)
55
```

Так можно «выполнить» скрипт из файла

```
exec(open(‘файл_на_питоне.py').read())
```

Вывод данных: функция print()

Функция **print** может выводить не только значения переменных, но и значения любых выражений.

При помощи функции **print** можно выводить значение не одного, а нескольких выражений, для этого нужно перечислить их через запятую:

```
a = 1
b = 2
print(a, '+', b, '=', a + b)
```

В данном случае будет напечатан текст 1 + 2 = 3

По «умолчанию» водимые значение разделяются одним пробелом.

Но такое поведение можно изменить:

- можно разделять выводимые значения любым другим символом,
- любой другой строкой,
- выводить их в отдельных строках или
- не разделять никак.

Для этого нужно функции **print** передать специальный именованный параметр, называемый **sep**, равный строке, используемый в качестве разделителя (**sep** — аббревиатура от слова separator, т.е. разделитель).

По умолчанию параметр **sep** равен строке из одного пробела и между значениями выводится пробел.

Чтобы использовать в качестве разделителя, например, символ двоеточия нужно передать параметр **sep**, равный строке ':'

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

Аналогично, для того, чтобы совсем убрать разделитель при выводе нужно передать параметр **sep**, равный пустой строке:

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

Для того, чтобы значения выводились с новой строке, нужно в качестве параметра **sep** передать строку, состоящую из специального символа новой строки, которая задается так:

```
print(a, b, sep = '\n')
```

Символ обратного слэша в текстовых строках является указанием на обозначение специального символа, в зависимости от того, какой символ записан после него.

Наиболее часто употребляется символ новой строки '\n'.

А для того, чтобы вставить в строку сам символ обратного слэша, нужно повторить его два раза: '\\'.

Вторым полезным именованным параметром функции **print** является параметр **end**, который указывает на то, что выводится после вывода всех значений, перечисленных в функции **print**.

По умолчанию параметр **end** равен **'\n'**, то есть следующий вывод будет происходить с новой строки.

Этот параметр также можно изменить, например, для того, чтобы убрать все дополнительные выводимые символы и «не переводить строку» можно вызывать функцию **print** так:

print(a, b, c, **sep = ''**, **end = ''**)

Целочисленная арифметика

Для целых чисел определены операции

+ - * ** % (остаток от деления)

Операция деления **/** для целых чисел возвращает значение типа **float**.

Также операция возведения в степень ** возвращает значение типа **float**, если показатель степени отрицательное число.

Но есть и специальная операция целочисленного деления, выполняющегося с отбрасыванием дробной части, которая обозначается **//**. Она возвращает целое число: целую часть частного.

Например:

```
>>> 17 // 3      # ====□ 5
>>> -17 // 3     # ====□ -6
```

Синтаксис условной инструкции

if ***Условие*****:**
    Блок инструкций 1
else:
    Блок инструкций 2

Внимание
«двоеточие»
«отступы»

Блок инструкций 1 будет выполнен, если Условие истинно. Если **Условие** ложно, будет выполнен *Блок инструкций 2.*

В условной инструкции может отсутствовать слово **else** и последующий блок. Такая инструкция называется неполным ветвлением.

Например:

```
if x < 0:
    x = -x
print(x)
```

*Для выделения **блока инструкций** (относящихся к инструкции if или else и др.) в языке Питон **используются отступы.***

Все инструкции, которые относятся к одному блоку, должны иметь равную величину отступа, то есть одинаковое число пробелов в начале строки.

Рекомендуется использовать отступ в 4 пробела и не рекомендуется использовать в качестве отступа символ табуляции.

Это одно из существенных отличий синтаксиса Питона от синтаксиса большинства языков, в которых блоки выделяются специальными словами, например, **begin**... **end** в Паскале или фигурными скобками { } в Си.

Вложенные условные инструкции

Внутри условных инструкций можно использовать любые инструкции языка Питон, в том числе и условную инструкцию.

Получаем вложенное ветвление – после одной развилки в ходе исполнения программы появляется другая развилка. *При этом вложенные блоки имеют больший размер отступа (например, 8 пробелов).*

Пример программы, которая по данным ненулевым числам x и y определяет, в какой из четвертей координатной плоскости находится точка (x,y):

```
x = int(input())
y = int(input())
if x > 0:
    if y > 0:                    # x>0, y>0
        print("Первая четверть")
    else:                        # x>0, y<0
        print("Четвертая четверть")
else:
    if y > 0:                    # x<0, y>0
        print("Вторая четверть")
    else:                        # x<0, y<0
        print("Третья четверть")
```

В этом примере использован **комментарии** – текст, который интерпретатор игнорирует.

Комментариями в Питоне является символ # и весь текст после этого символа до конца строки.

Операции сравнения

Как правило, в качестве проверяемого условия используется результат вычисления одного из следующих операций сравнения:

< Меньше — условие верно, если первый операнд меньше второго.

> Больше — условие верно, если первый операнд больше второго.

<= Меньше или равно.

>= Больше или равно.

== Равенство. Условие верно, если два операнда равны.

!= Неравенство. Условие верно, если два операнда неравны.

условие (x * x < 1000)
означает “значение x * x меньше 1000”

условие (2 * x != y) означает “удвоенное значение переменной x не равно значению переменной y”.

Операции сравнения в Питоне можно объединять в цепочки

в отличии от большинства других языков программирования, где для этого нужно использовать логические связки OR AND

например,

a == b == c

1 <= x <= 10

Тип данных bool

Операторы сравнения возвращают значения специального логического типа **bool**.

Значения логического типа могут принимать одно из двух значений: **True** (истина) или **False** (ложь).

Если преобразовать логическое **True** к типу **int**, то получится 1, а преобразование **False** даст 0.

При обратном преобразовании число 0 преобразуется в **False**, а любое ненулевое число в **True**.

При преобразовании **str** в **bool** пустая строка преобразовывается в **False**, а любая непустая строка в **True**.

Логические операции

В Питоне существуют стандартные логические операции: логическое И, логическое ИЛИ, логическое отрицание.

Логическое И является бинарной операцией **and**

Операция **and** возвращает **True** тогда и только тогда, когда оба операнда имеют значение **True**.

Логическое ИЛИ является бинарной операцией и возвращает **True** тогда и только тогда, когда хотя бы один операнд равен **True**. Операция “логическое ИЛИ” имеет вид **or**.

Логическое НЕ (отрицание) является унарной операцией и имеет вид **not**, за которым следует единственный операнд. Логическое НЕ возвращает **True**, если операнд равен **False** и наоборот.

Пример.

Проверим, что хотя бы одно из чисел a или b оканчивается на 0:

```
if a % 10 == 0 or b % 10 == 0:
```

Проверим, что число a — положительное, а b — неотрицательное:

```
if a > 0 and not (b < 0):
```

Или можно вместо **not** (b < 0) записать (b >= 0).

Каскадные условные инструкции

Пример программы, определяющий четверть координатной плоскости, можно переписать используя “каскадную“ последовательность операцией **if**... **elif**... **else**:

```
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 такой конструкции условия **if**, ..., **elif** проверяются по очереди, выполняется блок, соответствующий первому из истинных условий. Если все проверяемые условия ложны, то выполняется блок **else**, если он присутствует.

Цикл **for (**цикл с параметром)

В цикле for указывается переменная и множество значений, по которому будет пробегать переменная.

Множество значений может быть задано списком, кортежем, строкой или диапазоном.

Пример использования цикла, где в качестве множества значений используется ***кортеж***:

```
i = 1
for color in 'red', 'orange', 'yellow', 'green', 'cyan', 'blue', 'violet':
    print(i, '-й цвет радуги называется ', color, sep = '')
    i += 1
```

В списке значений могут быть выражения различных типов, например:

```
for i in 1, 2, 3, 'one', 'two', 'three':
    print(i)
```

При первых трех итерациях цикла переменная i будет принимать значение типа **int**, при последующих трех — типа **str**.

Функция range

Циклы **for обычно** используются для повторения какой-либо последовательности действий заданное число раз (**тела цикла**).

При этом выполнение операторов тела цикла происходит для значения **переменной цикла** от некоторого **начального значения** до **некоторого конечного** с заданным **шагом**.

Для «классической формы цикла со счетчиком» цикл **for используется с функцией range**:

```
for i in range(n):
    Тело цикла
```

В аргумент функции range(n) n может быть числовой константой, переменной или произвольным арифметическим выражением (например, 2 ** 10).

Если значение n равно нулю или отрицательное, то тело цикла не выполнится ни разу.

Если задать цикл таким образом:

```
for i in range(a, b):  # два аргумента
    Тело цикла
```

то индексная переменная i будет принимать значения от a до b - 1

первый параметр функции range(a, b) задает начальное значение индексной переменной, а второй параметр — конечное значение *увеличенное на единицу*.

b - значение, которая индексная переменная принимать не будет.

Если же a≥b, то цикл не будет выполнен ни разу.

Например сумма чисел от 1 до n:

```
sum = 0
for i in range(1, n + 1):
    sum += i
```

переменная i принимает значения 1, 2, ..., n, и значение переменной sum последовательно увеличивается на указанные значения.

Чтобы организовать цикл, в котором индексная переменная будет уменьшаться, необходимо использовать функцию **range** с тремя параметрами.

Первый параметр задает начальное значение индексной переменной, второй параметр — значение, до которого будет изменяться индексная переменная (не включая его!), а третий параметр — величину изменения индексной переменной.

Например, сделать цикл по всем нечетным числам от 1 до 99 можно при помощи функции **range**(1, 100, 2),

а сделать цикл по всем числам от 100 до 1 можно при помощи **range**(100, 0, -1).

цикл

for i **in range**(a, b, d):

при d > 0 задает значения индексной переменной i **=** a,

i **=** a + d,

i **=** a + 2 * d и так для всех значений,

для которых i < b.

range(1,20) =□ 1, 2, 3, ... 19

range(100,0,-2) =□ 100, 98, 96, 2, 1

range(10,1,1) = □ ????????

Функция enumerate()

Если нужно перебрать последовательность чисел, встроенная функция range() придёт на помощь.

Она генерирует арифметические прогрессии:

range(5) =>0 1 2 3 4

Указанный конец интервала никогда не включается в сгенерированный список;

вызов range(10) генерирует десять значений, которые являются подходящими индексами для элементов последовательности длины 10.

Можно указать другое начало интервала и другую, даже отрицательную, величину шага.

range(5, 10)

от 5 до 9

range(0, 10, 3)

0, 3, 6, 9

range(-10, -100, -30)

-10, -40, -70

Часто необходимо «обойти все элементы списка», например

a = ['Mary', 'had', 'a', 'little', 'lamb']

```
a = ['Mary', 'had', 'a',                \
                        'little', 'lamb']
for i in range(len(a)):
...print(i, a[i])


0 Mary
1 had
2 a
3 little
4 lamb
```

В большинстве таких случаев удобно использовать функцию enumerate().

Функция enumerate возвращает генератор, отдающий пары счётчик-элемент для элементов указанной последовательности.

Прототип функции

enumerate(sequence[, start=0]).

Параметры:

sequence – iterable Любая последовательность, итератор, или объект, поддерживающий итерирование.

start=0 -- int начально значение счётчика.

Результат: generator выдаёт пары счётчик-элемент.

Функция применяется в случаях, когда необходим счётчик количества элементов в последовательности и позволяет избавиться от необходимости инициализировать и обновлять отдельную переменную-счётчик:

Пусть задана последовательность

sequence = [1, 2, 7, 19]

вариант 1

```
idx = 0
  for item in sequence:
     print(idx)
     idx += 1
```

вариант 2

```
for idx, item in enumerate(sequence):
         print(idx)
```

```
a=['первый', 'второй', 'третий', \
        'четвертый', 'пятый']
for i in range(len(a)):
    print( i,  a[i])
print(10*'-')
i=0
for name in a:
    print(i, name)
    i+=1
print(10*'-')
for i, name in enumerate(a):
    print(i, name)
print(10*'-')
```

```
0 первый
1 второй
2 третий
3 четвертый
4 пятый
----------
0 первый
1 второй
2 третий
3 четвертый
4 пятый
----------
0 первый
1 второй
2 третий
3 четвертый
4 пятый
----------
```

Странные вещи начинают происходить при попытке вывода последовательности:

```
>>> print(range(10))
range(0, 10)
```

Во многих случаях объект, возвращаемый функцией range(), ведёт себя как список, но фактически им не является.

Этот объект возвращает по очереди элементы желаемой последовательности, когда вы проходите по нему в цикле, но на самом деле не создаёт списка, сохраняя таким образом пространство в памяти.

Такие объекты называются ***итерируемыми*** (iterable),

это все объекты, которые предназначаются для функций и конструкций, ожидающих от них поочерёдного предоставления элементов до тех пор, пока источник не иссякнет.

Оператор **for** является таким ***итератором*** **iterator**.

Функция **list()** тоже из их числа — она создаёт списки из *итерируемых* объектов:

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

В библиотеке питона есть и другие функции, которые возвращают и принимают ***итерируемые*** объекты в качестве параметров.

Действительные числа

Рассмотрим действительные числа, имеющие тип **float**.

Обратите внимание, что если вы хотите считать с клавиатуры действительное число, то результат, возвращаемый функцией input() необходимо преобразовывать к типу **float**:

```
x = float(  input()  )
```

Действительные (вещественные) числа представляются в виде чисел с десятичной точкой

Для записи очень больших или очень маленьких по модулю чисел используется так называемая запись “с плавающей точкой” (также называемая “научная” запись).

В этом случае число представляется в виде некоторой десятичной дроби, называемой мантиссой, умноженной на целочисленную степень десяти (порядок).

Например, расстояние от Земли до Солнца равно $1.496 \cdot 10^{11}$, а масса молекулы воды $2.99 \cdot 10^{-23}$.

Числа с плавающей точкой в программах на языке Питон, а также при вводе и выводе записываются в виде целой части, мантиссы, затем пишется буква e, затем пишется порядок.

Пробелы внутри этой записи не ставятся. Например, указанные выше константы можно записать в виде 1.496e11 и 2.99e-23.

Перед самим числом также может стоять знак минус.

Результатом операции деления / всегда является действительное число

Результатом операции // является целое число.

Преобразование действительных чисел к целому производится с округлением в сторону нуля

int(1.7) == 1 **int**(-1.7) == -1

Поддерживаются и комплексные числа, добавлением к мнимым частям суффикса j или J

Комплексные числа с ненулевым вещественным компонентом записываются в виде

(<вещественная_часть>+<мнимая_часть>j)

или могут быть созданы с помощью функции

complex(<вещественная_часть>, <мнимая_часть>).

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0, 1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

Библиотека math

Для проведения вычислений с действительными числами язык Питон содержит много дополнительных функций, собранных **в библиотеку** (модуль), которая называется **math**.

Для использования этих функций в начале программы необходимо ***подключить*** математическую библиотеку, что делается командой

```
import math
```

Функция от одного аргумента вызывается, например, так:
math.sin(x)
то есть явно указывается, что из модуля math используется функция sin.

Функция возвращает значение sin, которое можно вывести на экран, присвоить другой переменной или использовать в выражении:

```
y = math.sin(x)
print(math.sin(math.pi/2))
```

Другой способ использовать функции из библиотеки math, при котором не нужно будет при каждом использовании функции из модуля math указывать название этого модуля, выглядит так:

```
from math import *
 y = sin(x)
print(sin(pi/2))
```

Рассмотрим основные функций модуля math . Более подробное описание этих функций можно найти на сайте с документацией на Питон http://Рассмотрим основные функций модуля math . Более подробное описание этих функций можно найти на сайте с документацией на Питон http://docs.python.org/py3k/library/math.html)

*Некоторые из перечисленных функций (**int**, **round**, **abs**) являются стандартными и не*

Функция Описание

Округление

`int(x)` Округляет число в сторону нуля. Это стандартная функция, для ее использования не нужно подключать модуль `math`.

`round(x)` Округляет число до ближайшего целого. Если дробная часть числа равна 0.5, то число округляется до ближайшего четного числа.

`round(x, n)` Округляет число `x` до `n` знаков после точки. Это стандартная функция, для ее использования не нужно подключать модуль `math`.

`floor(x)` Округляет число вниз (“пол”), при этом

```
floor(1.5) == 1,   floor(-1.5) == -2
```

`ceil(x)` Округляет число вверх (“потолок”), при этом

```
ceil(1.5) == 2,   ceil(-1.5) == -1
```

`trunc(x)` Округление в сторону нуля (так же, как функция `int`).

`abs(x)` Модуль (абсолютная величина). Это – стандартная функция.

`fabs(x)` Модуль (абсолютная величина). Эта функция всегда возвращает значение типа `float`.

Функция Описание

Корни, степени, логарифмы

sqrt(x) Квадратный корень.
Использование: sqrt(x)

pow(a, b) Возведение в степень, возвращает a^b
Использование: pow(a,b)

exp(x) Экспонента, возвращает e^x.
Использование: exp(x)

log(x) Натуральный логарифм.
При вызове в виде log(x, b) возвращает логарифм по основанию b.

log10(x) Десятичный логарифм

math.e Основание натуральных логарифмов

Тригонометрия

`sin(x)` **Синус угла, задаваемого в радианах**

`cos(x)` **Косинус угла, задаваемого в радианах**

`tan(x)` **Тангенс угла, задаваемого в радианах**

`asin(x)` **Арксинус, возвращает значение в рад**

`acos(x)` **Арккосинус, возвращает значение в рад**

`atan(x)` **Арктангенс, возвращает значение в радианах**

`atan2(y, x)` **Полярный угол (в радианах) точки с координатами (x, y).**

`hypot(a,b)` **Длина гипотенузы прямоугольного треугольника с катетами a и b.**

`degrees(x)` **Преобразует угол, заданный в радианах, в градусы.**

`radians(x)` **Преобразует угол, заданный в градусах, в радианы.**

`math.pi` **Константа** π

Строки

Строка считывается со стандартного ввода функцией input() или **задается константой** – набором букв в кавычках. **Строки могут быть заключены в одинарные или двойные кавычки**

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

Интерпретатор выводит результаты операций над строками тем же способом, каким они были введены: обрамляя в кавычки, и, кроме того, **экранируя обратными слэшами** внутренние кавычки и другие *забавные* символы — для того, чтобы отобразить точное значение.

Строка заключается в двойные кавычки, если строка содержит одинарные кавычки и ни одной двойной, иначе она заключается в одинарные кавычки.

Функция print() предоставляет более читаемый вывод.

Строковые литералы могут быть *разнесены на несколько строк* различными способами. Могут быть использованы ***продолжающие строки*, с обратным слэшем в качестве последнего символа строки,** сообщающим о том, что следующая строка есть продолжение текущей:

```
hello = "This is a rather long string containing\n\
    several lines of text just as you would do in C.\n\
        Note that whitespace at the beginning of the line
    is\ significant."
 print(hello)
```

print(hello)

Обратите внимание, что новые строки нужно подключать в строку через \n

новая строка, за которой следут обратный слэш — не обрабатывается. Запуск примера выведет следующее:

This is a rather long string containing
several lines of text just as you would do in C.

Note that whitespace at the beginning of the line is significant.

Можно объявить строковой литерал *сырым* (raw) — символы \n не будут конвертированы в новые строки, но и обратный слэш в конце строки, и символ новой строки в исходном коде — будут добавлены в строку в виде данных. Следовательно, код из примера:

```
hello = r"This is a rather long string containing\n\
    several lines of text much as you would do in C."
```

print(hello) выведет:

```
This is a rather long string containing\n\ several
    lines of text much as you would do in C.
```

Строки могут быть обрамлены совпадающей парой тройных кавычек: """ или '''.

Окончания строк не нужно завершать тройными кавычками - при они при этом этом будут включены в строку автоматически.

```
print("""
    Usage: thingy [OPTIONS]
    -h                    Display this usage message
    -h  hostname    Hostname to connect to
""")
```

выводит в результате следующее:

```
Usage: thingy [OPTIONS]
    -h                    Display this usage message
    -h hostname     Hostname to connect to
```

Два строковых литерала, расположенные друг за другом, автоматически конкатенируются;

word = 'Help' + 'A'

word = 'Help' 'A';

это работает только с двумя литералами, а не с произвольными выражениями, содержащими строки.

```
>>> 'str' 'ing'                    #  <-  Так – верно
'string'
>>> 'str'.strip() + 'ing'   #  <-  Так – верно
'string'
>>> 'str'.strip() 'ing'     #  <-  Так – не верно
  File "<stdin>", line 1, in ?
    'str'.strip() 'ing'
                  ^
SyntaxError: invalid syntax
```

Для строк определена операция сложения (конкатенации), также определена операция умножения строки на число.

Строка состоит из последовательности символов. Узнать количество символов (длину строки) можно при помощи функции **len**:

```
>>> S = 'Hello'
>>> print(len(S))
5
```

Срезы (slices)

Срез (slice) — извлечение из данной строки одного символа или некоторого фрагмента подстроки или подпоследовательности.

Есть три формы срезов.

1 Самая простая форма среза: ***взятие одного символа строки***, а именно, **S[i] — это срез, состоящий из одного символа**, который имеет номер i, при этом считая, что ***нумерация начинается с числа 0***.

То есть если S='Hello', то

S[0]=='H', S[1]=='e', S[2]=='l', S[3]=='l', S[4]=='o'.

Номера символов в строке (а также в других структурах данных: списках, кортежах) называются *индексом.*

Если указать отрицательное значение индекса, то номер будет отсчитываться с конца, начиная с номера -1.

***То есть* S[-1]=='o', S[-2]=='l', S[-3]=='l', S[-4]=='e', S[-5]=='H'.**

Или в виде таблицы:

Строка S	H	e	l	l	o
Индекс	S[0]	S[1]	S[2]	S[3]	S[4]
Индекс	S[-5]	S[-4]	S[-3]	S[-2]	S[-1]

Если же номер символа в срезе строки S больше либо равен len(S), или меньше, чем -len(S), то при обращении к этому символу строки произойдет ошибка ***IndexError: string index out of range****.*

2. **Срез с двумя параметрами: S[a:b]**
возвращает подстроку из b-a символов, начиная с символа с индексом a, до символа с индексом b, не включая его.

Например, S[1:4]=='ell', то же самое получится если написать S[-4:-1].

Можно использовать как положительные, так и отрицательные индексы в одном срезе,

например, S[1:-1] — это строка без первого и последнего символа (срез начинается с символа с индексом 1 и заканчивается индексом -1, не включая его).

При использовании такой формы среза ошибки ***IndexError*** *никогда не возникает.*

Например, срез S[1:5] вернет строку 'ello', таким же будет результат, если сделать второй индекс очень большим, например, S[1:100] (если в строке не более 100 символов).

Если опустить второй параметр (но поставить двоеточие), то срез берется от k-го символа до конца строки S[k:]

Чтобы удалить из строки первый символ (его индекс равен 0) можно взять срез, начиная с символа с индексом 1 S[1:]

Если опустить первый параметр, то срез берется от начала строки до k-го символа S[: k].

Удалить из строки последний символ можно при помощи среза S[:-1].

Срез S[:] совпадает с самой строкой S.

3. Если задать **срез с тремя параметрами** S[a:b:d], то третий параметр задает шаг, как в случае с функцией **range**

S[a:b:d] - будут взяты символы с индексами a, a+d, a+2*d и т.д.

При задании значения третьего параметра, равному 2, в срез попадет каждый второй символ, а если взять значение среза, равное -1, то символы будут идти в обратном порядке.

s='0123456789‘

s[5:0:-1] --□ '54321'

s[5::-1] --□ '543210'

Методы

Метод - это функция, применяемая к объекту, в данном случае - к строке.

Метод вызывается в виде **Имя_объекта.Имя_метода (параметры)**

Например, **S.find("e")** — это применение к строке **S** метода **find** с одним параметром **"e".**

Метод find и rfind

Объект.find(подстрока)

Метод **find** находит в данной строке (к которой применяется метод) подстроку, которая передается в качестве параметра («поиск с начала строки»).

Функция возвращает индекс первого вхождения искомой подстроки.

Если же подстрока не найдена, то метод возвращает значение -1

```
>>> S = 'Hello'
>>> print(S.find('e'))
1
>>> print(S.find('ll'))
2
>>> print(S.find('L'))
-1
```

Аналогично, метод **rfind** возвращает индекс последнего вхождения данной строки (“поиск справа”):

```
>>> S = 'Hello'
>>> print(S.find('l'))
2
>>> print(S.rfind('l'))
3
```

Если вызвать метод find с тремя параметрами S.find(T, a, b), то поиск будет осуществляться в срезе S[a:b].

Если указать только два параметра S.find(T, a), то поиск будет осуществляться в срезе S[a:] - начиная с символа с индексом a и до конца строки.

Метод S.find(T, a, b) возвращает индекс в строке S, а не индекс относительно среза.

Метод replace

Метод **replace** заменяет все вхождения одной строки на другую.

Формат:

S.replace(old, new)

заменить в строке S все вхождения подстроки old на подстроку new.

Пример:

```
>>> 'Hello'.replace('l', 'L')
'HeLLo'
```

Если методу **replace** задать еще один параметр: **S.replace(old, new, count),**

то заменены будут не все вхождения, а только не больше, чем ***первые count*** из них.

```
>>> 'Abrakadabra'.replace('a', 'A', 2)
'AbrAkAdabra'
```

Метод count

Подсчитывает количество вхождений одной строки в другую строку. Простейшая форма вызова

S.count(T)

возвращает число вхождений строки T внутри строки S.

При этом подсчитываются только непересекающиеся вхождения, например:

```
>>> 'Abracadabra'.count('a')
4
>>> ('a' * 100000).count('aa')
50000
```

При указании трех параметров

S.count(T, a, b),

будет выполнен подсчет числа вхождений строки T в срез S[a:b].

Цикл while

Цикл **while** (“пока”) позволяет выполнить одну и ту же последовательность действий, пока проверяемое условие истинно.

Условие проверяется до выполнения тела цикла.

Как правило, цикл **while** используется, когда невозможно определить точное значение количества итераций тела цикла.

Синтаксис цикла **while** в простейшем случае выглядит так:

```
while условие:
    блок инструкций
```

При выполнении цикла **while** сначала проверяется условие.

Если оно ложно, то выполнение цикла прекращается и управление передается на следующую инструкцию после тела цикла **while**.

Если условие истинно, то выполняются инструкции тела цикла, после чего условие проверяется снова

Так продолжается до тех пор, пока условие будет истинно.

Как только условие станет ложно, работа цикла завершится и управление передастся следующей инструкции после цикла.

фрагмент программы выведет квадраты всех целых чисел от 1 до 10 (цикл **while** может заменять цикл **for** ... **in range**(...)):

```
i = 1
while i <= 10:
    print(i*i)
    i += 1
```

В этом примере переменная i изменяется от 1 до 10. Эта переменная, значение которой меняется с каждым новым проходом цикла (итерацией), выполняет роль счетчика.

Заметим, что после выполнения этого фрагмента значение переменной i будет равно 11, т.к. при i==11 условие i<=10 впервые перестанет выполняться.

пример использования цикла **while** для определения количества цифр натурального числа n:

```
n = int(input())
length = 0
while n > 0:
    n //= 10
    length += 1
```

В этом цикле отбрасываем по одной цифре числа, начиная с конца, что эквивалентно целочисленному делению на 10 (n //= 10), при этом считаем в переменной length, сколько раз это было сделано.

В языке Питон есть и другой способ решения этой задачи: **length = len(str(i)).**

Инструкции управления циклом

После тела цикла можно написать слово else: и после него блок операций

Этот блок будет выполнен один раз после окончания цикла, когда проверяемое условие станет ложным:

```
i = 1
while i <= 10:
    print(i)
    i += 1
else:
    print('Цикл окончен, i =', i)
```

Казалось бы, никакого смысла в этом нет, ведь эту же инструкцию можно просто написать после окончания цикла.

Смысл в блоке else: появляется только вместе с **инструкцией break**

использование **break** внутри цикла приводит к немедленному прекращению итераций цикла, ***и при этом не исполняется ветка*** **else**.

break осмысленно вызывается только из **if**, то есть обычно **break** выполняться только при выполнении какого-то специального.

Другая **инструкция управления циклом — continue** **-** продолжение итераций цикла.

Если эта инструкция встречается где-то посередине тела цикла, то ***пропускаются все оставшиеся инструкции до конца тела цикла, и исполнение цикла продолжается со следующей итерации***.

Инструкции **break**, **continue** и ветка **else**: можно использовать и внутри цикла **for**.

Тем не менее, увлечение инструкциями **break** и **continue** не поощряется, если можно рекомендуется обойтись без их использования.

Вот типичный пример использования инструкции **break**

```
length=0
while True:
    length += 1
    n //= 10
    if n == 0:
        break
```

Оператор pass не делает ничего.

Он может использоваться когда синтаксически требуется присутствие оператора, но от программы не требуется действий. Например:

```
>>> while True:
...     pass  # Ожидание прерывания с клавиатуры (Ctrl+C) в режиме занятости...
```

Этот оператор часто используется для создания, к примеру *исключений* (exceptions), или для игнорирования нежелательных исключений:

```
>>> try:
...     import audioop...
except ImportError:
...     pass
...
```

Другой вариант: **pass** может применяться в качестве заглушки для тела функции или условия при создании нового кода, позволяя вам сохранить абстрактный взгляд на вещи.

С другой стороны, оператор **pass** игнорируется без каких-либо сигналов и лучшим выбором было бы породить исключение NotImplementedError:

```
>>> def initlog(*args):
...     raise NotImplementedError   # Открыть файл для
                                    # логгинга,
                                    # если он ещё
                                    # не открыт
...     if not logfp:
...         raise NotImplementedError  # Настроить
                                    # заглушку для
                                    # логгинга
...     raise NotImplementedError(
                    'Обработчик инициализации лога вызовов')
```

... Если бы здесь использовались операторы **pass**, а позже вы бы запускали тесты, они могли бы «упасть» без указания причины.

Использование NotImplementedError принуждает этот код породить исключение, сообщая вам конкретное место, где присутствует незавершённый код.

Функции

Рассмотрим задачу вычисления числа сочетаний из n элементов по k, для чего необходимо вычисление факториалов трех величин: n, k и n-k.

Для этого можно сделать три цикла, что приводит к увеличению размера программы за счет трехкратного повторения похожего кода.

Вместо этого лучше сделать одну функцию, вычисляющую факториал любого данного числа n и трижды использовать эту функцию в своей программе.

```
def factorial(n):
    f = 1
    for i in range(2, n + 1):
        f *= i
    return f
```

Этот текст должен идти в начале программы, вернее, до того места, где мы захотим воспользоваться функцией factorial.

Первая строчка этого примера является описанием нашей функции. Зарезервированное слово **def** предваряет *определение* функции. За ним должны следовать «имя функции» -

factorial — идентификатор, то есть имя нашей функции.

После идентификатора в круглых скобках идет список параметров, которые получает наша функция.

Список состоит из перечисленных через запятую идентификаторов параметров.

В нашем случае список состоит из одной величины n. В конце строки ставится двоеточие.

Далее идет **тело функции**, оформленное в виде блока, то есть с отступом.

Первым выражением в теле функции может быть строковой литерал — этот литерал является строкой документации функции, или *док-строкой* (docstring). (не обязательный элемент).

Существуют инструменты, которые используют *док-строки* для того, чтобы сгенерировать печатную или онлайн - документацию или чтобы позволить пользователю перемещаться по коду интерактивно;

добавление строк документации в ваш код - это хорошая практика, постарайтесь к ней привыкнуть.

Внутри функции вычисляется значение факториала числа n и оно сохраняется в переменной f.

Функция завершается инструкцией **return** f, которая завершает работу функции и возвращает значение переменной f.

Инструкция **return** может встречаться в произвольном месте функции, ее исполнение завершает работу функции и возвращает указанное значение в место вызова.

Если функция не возвращает значения, то инструкция **return** используется без возвращаемого значения, также

В функциях, не возвращающих значения, инструкция **return** может отсутствовать.

Теперь можно использовать функцию несколько раз.

В этом примере трижды вызываем функцию **factorial** для вычисления трех факториалов: **factorial**(n), **factorial**(k), **factorial**(n-k).

```
n = int(input())
k = int(input())
print( factorial(n)  //                          \
              (factorial(k)*factorial(n-k)) )
```

Можно, например, объявить функцию **binomial**, которая принимает два целочисленных параметра n и k и вычисляет число сочетаний из n по k:

```
def binomial(n, k):
    return factorial(n)//(factorial(k) *      \
                          factorial(n - k))
```

Тогда в основной программе можем вызвать функцию binomial для нахождения числа сочетаний из n по k :

```
print ( binomial(n, k)  )
```

Рассмотри задачу нахождения наибольшего из двух или трех чисел. Функцию нахождения максимума из двух чисел можно написать так:

```
def max(a, b):
    if a > b:
        return a
    else:
        return b
```

Теперь можем реализовать функцию **max3**, находящую максимум трех чисел:

```
def max3(a, b, c):
    return max(max(a, b), c)
```

Функция **max3** дважды вызывает функцию **max** для двух чисел: сначала, чтобы найти максимум из a и b, потом чтобы найти максимум из этой величины и c.

Локальные и глобальные переменные

Внутри функции можно использовать переменные, объявленные вне этой функции

```
def f():
    print(a)
a = 1
f()
```

Здесь переменной a присваивается значение 1, и функция f печатает это значение, несмотря на то, что в функции f эта переменная f не инициализируется.

Но в момент вызова функции f переменной **a** уже присвоено значение, поэтому функция f может вывести его на экран.

Такие **переменные - объявленные вне функции, но доступные внутри функции, называются глобальными**.

Но если *инициализировать какую-то переменную внутри функции*, использовать эту переменную вне функции не удастся.

Например:

```
def f():
    a = 1
f()
print(a)
```

Получим

NameError: name 'a' is not defined.

Такие *переменные, объявленные внутри функции, называются локальными.*

Эти переменные становятся недоступными после выхода из функции.

Интересным получится результат, если попробовать изменить значение глобальной переменной внутри функции:

```
def f():
    a = 1
    print(a)
a = 0
f()
print(a)
```

Будут выведены числа 1 и 0.

То есть несмотря на то, что значение переменной a изменилось внутри функции, то вне функции оно осталось прежним!

Это сделано в целях “защиты” глобальных переменных от случайного изменения из функции

например, если функция будет вызвана из цикла по переменной i, а в этой функции будет использована переменная i также для организации цикла, то эти переменные должны быть различными)

Если внутри функции модифицируется значение некоторой переменной, то переменная с таким именем становится *локальной переменной*, и ее модификация не приведет к изменению глобальной переменной с таким же именем.

интерпретатор Питон считает переменную локальной, если внутри нее есть хотя бы одна инструкция, модифицирующая значение переменной.

Это может быть оператор =, **+=** и т.д., или использование этой переменной в качестве параметра цикла f**or**.

При этом даже если инструкция, модицифицирующая переменную **никогда не будет выполнена**, интерпретатор это проверить не может, и переменная **все равно считается локальной**

Пример:

```
def f():
    print (a)
    if False:
        a = 0
a = 1
f()
```

Возникает ошибка:

UnboundLocalError: local variable 'a' referenced before assignment.

В функции f идентификатор **a** становится локальной переменной, т.к. в функции есть команда, модифицирующая переменную a, пусть даже никогда и не выполняющийся (но интерпретатор не может это отследить).

Поэтому вывод переменной **a** приводит ***к обращению к неинициализированной локальной переменной***.

Чтобы функция могла изменить значение глобальной переменной, необходимо объявить эту переменную внутри функции, как глобальную, при помощи ключевого слова **global**:

```
def f():
    global a
    a = 1
    print (a)
a = 0
f()
print(a)
```

В этом примере на экран будет выведено 1 и 1, так как переменная **a** объявлена как глобальная

Изменение **a** внутри функции приводит к тому, что и вне функции переменная будет доступна и равна значению, присвоенному в функции

Не рекомендуется изменять значения глобальных переменных внутри функции.

Если функция должна поменять какую-то переменную, то как правило, это лучше сделать, через значение, возвращаемое функцией.

Если нужно, чтобы функция вернула не одно значение, а два или более, то для этого **функция может вернуть кортеж** из двух или нескольких значений:

return (a, b)

Результат вызова функции нужно присваивать кортежу:

(n, m) **=** f(a, b)

Более подробнее о функциях

Исполнение функции приводит к созданию новой ***таблицы символов***, использующейся для хранения локальных переменных функции.

Если быть более точными, все присваивания переменных в функции сохраняют значение в локальной таблице символов;

при обнаружении ссылки на переменную, в первую очередь просматривается локальная таблица символов, затем локальная таблица символов для окружающих функций, затем глобальная таблица символов и, наконец, таблица встроенных имён.

Таким образом, глобальным переменным невозможно прямо присвоить значения внутри функций (если они конечно не упомянуты в операторе **global**) несмотря на то, что ссылки на них могут использоваться.

Фактические параметры при вызове функции помещаются в локальную таблицу символов вызванной функции; в результате аргументы передаются через ***вызов по значению*** (call by value) (где значение - это всегда *ссылка* (reference) на объект, а не значение его самого).

Если одна функция вызывает другую — то для этого вызова создается новая локальная таблица символов.

создадим функцию, которая выводит числа Фибоначчи до некоторого предела:

```
>>> def fib(n):    # вывести числа Фибоначчи
                   # меньшие (вплоть до) n...
    ""
    Выводит ряд Фибоначчи, ограниченный n.
    """
...     a, b = 0, 1
...     while b < n:
...         print(b, end=' ')
...         a, b = b, a+b
...     print()...
>>> # Теперь вызовем определенную нами функцию:...
    fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

При определении функции её имя также помещается в текущую таблицу символов.

Тип значения, связанного с именем функции, распознается интерпретатором как функция, определённая пользователем (user-defined function).

Само значение может быть присвоено другому имени, которое затем может также использоваться в качестве функции.

Эта система работает в виде основного механизма переименования:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
1 1 2 3 5 8 13 21 34 55 89
```

В терминах других языках программирования, fib — это не функция, а процедура, поскольку не возвращает никакого значения.

На самом деле, даже функции без ключевого слова **return** возвращают значение, хотя и более скучное.

Такое значение именуется **None** (это встроенное имя).

Вывод значения **None** обычно подавляется в интерактивном режиме интерпретатора, если оно оказывается единственным значением, которое нужно вывести.

Вы можете проследить за этим процессом, если действительно хотите, используя функцию print():

```
>>> fib(0)
>>> print(fib(0))
None
```

Довольно легко написать функцию, которая возвращает список чисел из ряда Фибоначчи, вместо того, чтобы выводить их:

```
def fib2(n):
    """
    Возвращает список чисел ряда
        Фибоначчи,ограниченный n.
    """
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)    # см. ниже
        a, b = b, a+b
    return result
>>> z=fib2(100)     # вызываем
>>> print(z)             # выводим результат
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Оператор **return** завершает выполнение функции, возвращая некоторое значение.

Оператор **return** без аргумента возвращает **None**.

Достижение конца функции также возвращает **None**.

Выражение result.append(b) вызывает метод append объекта-списка result.

Метод — это функция, которая "принадлежит" объекту и указывается через выражение вида
obj.methodname,
где obj — некоторый объект (может быть выражением), а methodname — имя метода, присущий объекту данного типа.

Различные типы определяют различные методы. Методы разных типов могут иметь одинаковые имена, не вызывая неопределённостей.

Метод append(), показанный в примере, определён для объектов типа список. Он добавляет в конец списка новый элемент.

В данном примере это действие эквивалентно выражению result = result + [b], но более эффективно.

Рекурсия

```
#Эпиграф(ошибка рекурсии…):
def ShortStory():
    print("У попа была собака, он ее
  любил.")
    print("Она съела кусок мяса, он ее
  убил,")
    print("В землю закопал и надпись
  написал:")
    ShortStory()
```

примере функции вычисления факториала.

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

Подобный прием - вызов функцией самой себя, называется **рекурсией**, а сама функция называется рекурсивной.

Рекурсивные функции являются мощным механизмом в программировании.

К сожалению, они всегда не эффективны

Также часто использование рекурсии приводит к ошибкам – возникновение бесконечной рекурсии - цепочка вызовов функций никогда не завершается и продолжается, пока не кончится свободная память

Две наиболее распространенные причины для бесконечной рекурсии:

- Неправильное оформление выхода из рекурсии.
- Рекурсивный вызов с неправильными параметрами.

Поэтому при разработке рекурсивной функции необходимо прежде всего оформлять условия завершения рекурсии

Подробнее об определении функций

Есть возможность определять функции с переменным количеством параметров. Для этого существует три формы, которые также можно использовать совместно.

Значения аргументов по умолчанию

Наиболее полезной формой является задание значений по умолчанию для одного или более параметров.

Таким образом создаётся функция, которая может быть вызвана с меньшим количеством параметров, чем в её определении

при этом неуказанные при вызове параметры примут данные в определении функции значения

Например:

```
def ask_ok(prompt,retries=4,\
           complaint='Yes or no, please!'):
      while True:
          ok = input(prompt)
          if ok in ('y', 'ye', 'yes'):
             return True
          if ok in ('n', 'no', 'nop', 'nope'):
             return False
          retries = retries - 1
          if retries < 0:
              raise IOError('refusenik user')
       print(complaint)
```

Эта функция может быть вызвана, например, так: ask_ok('Do you really want to quit?') или так: ask_ok('OK to overwrite the file?', 2).

пример также знакомит вас

с зарезервированным словом **in**. Посредством его можно проверить, содержит ли последовательность определённое значение или нет

raise `IOError – «выбрасывает исключение» IOError`

Значения по умолчанию вычисляются в месте определения функции, в *определяющей* области, поэтому код

```
i = 5
def f(arg=i):
    print(arg)
……..
i = 6
f()
```

выведет 5

Важное предупреждение: **Значение по умолчанию вычисляется лишь единожды.** Это особенно важно помнить, когда значением по умолчанию, например, является изменяемый объект, такой как список, словарь (dictionary).

Например, следующая функция накапливает переданные ей параметры:

```
def f(a, L=[]):
        L.append(a)
        return L
print( f(1))
print(f(2))
print(f(3))
```

Она выведет

```
[1]
[1, 2]
[1, 2, 3]
```

Если вы не хотите, чтобы значение по умолчанию распределялось между последовательными вызовами, вместо предыдущего варианта вы можете использовать:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

Именованные параметры

Функции также могут быть вызваны с использованием именованных параметров (keyword arguments) в форме "***имя = значение***". Например, нижеприведённая функция:

```
def parrot(voltage, state='a stiff', action='voom',
                                        type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

могла бы быть вызвана любым из следующих способов:

```
parrot(1000)
parrot(action='VOOOOOM', voltage=1000000)
parrot('a thousand', state='pushing up the daisies')
parrot('a million', 'bereft of life', 'jump')
```

эти случаи неверные:

```
# пропущен требуемый  аргумент
parrot()
 # позиционный параметр вслед за
   именованным
parrot(voltage=5.0, 'dead')
# повторное значение параметра
parrot(110, voltage=220)
 # неизвестное имя параметра
parrot(actor='John Cleese')
```

В общем случае, список параметров должен содержать любое количество позиционных (positional) параметров, за которыми может следовать любое количество именованных, и при этом имена аргументов выбираются из формальных параметров.

Неважно, имеет формальный параметр значение по умолчанию или нет.

Ни один из аргументов не может получать значение более чем один раз

Имена формальных параметров, совпадающие с именами позиционных параметров, не могут использоваться в качестве именующих в одном и том же вызове.

Вот пример, завершающийся неудачей по причине описанного ограничения:

```
def function(a):
...     pass...
>>> function(0, a=0)
Traceback (most recent call last):  File "<stdin>", line 1, in ?TypeError:
    function() got multiple values for keyword argument 'a'
```

Если в определении функции присутствует завершающий параметр в виде ***имя**, он получит в качестве значения **словарь**, содержащий **все именованные параметры и их значения**, исключая те, которые соответствуют присутствующим формальным параметрам.

Можно совместить эту особенность с поддержкой формального параметра в формате ***имя** *(который получает кортеж **tuple**), содержащий все позиционные параметры*, следующие за списком формальных параметров.

Параметр в формате **имя* должен описываться перед параметром в формате ***имя*.

Пример

```
def cs(kind, *arguments, **keywords):
    print("-- kind=", kind)
    for arg in arguments:
        print(arg)
    print("-" * 40)
    keys = sorted(keywords.keys())
    for kw in keys:
        print(kw, ":", keywords[kw])


cs("aaaaaa",  '111', '2222', '33333',  zit=5555,
                                p1=1, p2=2, p3='3333')
```

```
-- kind= aaaaaa
111
2222
33333
----------
p1 : 1
p2 : 2
p3 : 3333
zit : 5555
>>>
```

Список имён (ключей) именованных параметров (keys) создается ***посредством сортировки*** содержимого списка ключей keys() словаря keywords. Если бы этого не было сделано, порядок вывода параметров был бы произволен.

Списки параметров произвольной длины

Есть возможность указать что функция может быть вызвана с произвольным числом аргументов.

При этом сами параметры будут представлены кортежем.

Переменное количество параметров могут предварять ноль или более обычных параметров.

Обычно параметры неизвестного заранее количества (variadic) указываются последними в списке формальных параметров, поскольку включают в себя все остальные переданные в функцию параметры.

Все формальные параметры, которые следуют за параметром *args, должны быть только именованными, то есть, они могут быть заданы только по имени (в отличие от позиционных параметров).

```
def concat(*args,sep="/"):
    for w in args:
        print(w)
    return sep.join(args)
################################
s1=concat("111", '222', '333')
print('rv=',s1)
s2=concat("111", '222', '333', sep=".")
print('rv=',s2)
```

```
111
222
333
rv= 111/222/333
111
222
333
rv= 111.222.333
```

Распаковка списков параметров

Обратная ситуация возникает когда параметры уже содержатся в списке или в кортеже, но должны быть распакованы для вызова функции, требующей отдельных позиционных параметров.

Например, встроенная функция range() ожидает отдельные параметры *start* и *stop* соответственно.

Если они не доступны раздельно, для распаковки аргументов из списка или кортежа в вызове функции используйте *-синтаксис:

```
>>> list(range(3, 6))  #  обычный вызов с
                          #  отдельными параметрами
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))    # вызов с распакованными из
                             #   списка параметрами
[3, 4, 5]
```

словари могут получать именованные параметры через **-синтаксис:

```
def par(volt, state='stiff', act='go'):
  print("action=", act, end=' ')
  print("volt=",volt,end=' ')
  print("state=", state)

d = {"volt": "11111", "state": "222222", "act": "zzz"}
par(**d)
action= zzz volt= 11111 state= 222222
d1 = {"volt": "11111"}
par(**d1)
action= go volt= 11111 state= stiff
```

Модель lambda

Используя зарезервированное слово **lambda** можно создать небольшую ***безымянную функцию***.

Например, функцию, которая возвращает сумму двух своих аргументов, можно записать так:

lambda a, b : a+b

Формы **lambda** могут быть использованы в любом месте где требуется объект функции.

При этом они синтаксически ограничены одним выражением.Семантически, они лишь «синтаксический сахар» для обычного определения функции. Как и определения вложенных функций, **lambda**-формы могут ссылаться на переменные из содержащей их области видимости:

```
def m_i(n):
    return lambda x: x + n

>>> f=m_i(54)
>>> print(f)
<function m_i.<locals>.<lambda> at 0x00000198277E81F0>
>>> f(0)
54
>>> f(1)
55
>>>
```

Списки

Большинство программ работает не с отдельными переменными, а с набором переменных. Во многих задачах нужно иметь возможность работать со всеми элементами последовательности. В большинстве языков программирования для этого используется объект программы, называемый **массивом**.

В Питоне «формально» массивы как объект программы существует, но имеется возможность (а некоторые авторы рекомендуют) использовать «обобщенную» структуру данных - **список**

Список представляет собой последовательность из n элементов, пронумерованных целыми числами от 0 до (n-1)

Список можно задать перечислением элементов списка в квадратных скобках, например:

```
Primes = [2, 3, 5, 7, 11, 13]
Rainbow = ['Red', 'Orange', 'Yellow', 'Green', 'Blue',  \
                                                'Indigo', 'Violet']
```

В списке Primes 6 элементов:

Primes[0] == 2, Primes[1] == 3, Primes[2] == 5,

Primes[3] == 7, Primes[4] == 11, Primes[5] == 13.

Список Rainbow состоит из 7 элементов, каждый из которых является строкой.

Элементы списка можно индексировать отрицательными числами с конца, например:

Primes[-1] == 13, Primes[-6] == 2.

Длину списка, то есть количество элементов в нем, можно узнать при помощи функции **len**, например:

len(Primes) == 6.

В Питоне предусмотрено несколько способов создания списка и работы с его элементами

Можно создать пустой список (не содержащий элементов, длины 0). **Пустой список обозначается как [ ]**

В конец списка можно добавлять элементы при помощи метода **append**.

Например:

программа вводит количество элементов в списке n, а потом n элементов списка по одному вводятся и из них формируется список:

```
A = []
for i in range( int(input() ):
    A.append(int(input())
```

В этом примере создается пустой список, далее считывается количество элементов в списке, затем по одному считываются элементы списка и добавляются в его конец.

Для списков целиком определены следующие основные операции:

конкатенация списков “+” - добавление одного списка в конец другого

повторение списков “*” «**умножение списка на число**».

Например:

A **=** [1, 2, 3]

B **=** [4, 5]

C **=** A + B

D **=** B * 3

список C будет равен [1, 2, 3, 4, 5]

список D будет равен [4, 5, 4, 5, 4, 5].

Например, можно по другому организовать процесс считывания списков:

сначала считать размер списка и создать список из нужного числа элементов,

затем организовать цикл по переменной i начиная с числа 0 и в теле цикла считывать и «определять» i-й элемент списка:

```
A = [0] * int(input())
for i in range(len(A)):
    A[i] = int(input())
```

Вывести элементы списка A можно одной инструкцией **print**(A), при этом будут выведены квадратные скобки вокруг элементов списка и запятые между элементами списка.

Такой вывод неудобен, чаще требуется просто вывести все элементы списка в одну строку или по одному элементу в строке.

Рассмотрим два примера:

```
for i in range(len(A)):
    print(A[i])
```

Здесь в цикле по индексу элемента i выводится элемент списка с индексом i.

```
for elem in A:
    print(elem, end = ' ')
print()
```

В этом примере элементы списка выводятся в одну строку, разделенные пробелом, при этом в цикле меняется не индекс элемента списка, а само значение переменной.

В цикле

```
for elem in ['red', 'green', 'blue']
```

переменная elem будет последовательно принимать значения 'red', 'green', 'blue'

Методы split и join

Элементы списка могут вводиться по одному в строке. Строку можно считать функцией **input**().

После этого можно использовать метод строки **split**, возвращающий список строк, «разрезав» исходную строку на части по пробелам.

Пример:

```
A = input().split()
```

Если при запуске этой программы ввести строку 1 2 3, то список A будет равен ['1', '2', '3'].

Обратите внимание, что список будет состоять из строк, а не из чисел.

Если хочется получить список именно из чисел, то можно затем элементы списка по одному преобразовать в числа:

```
for i in range(len(A)):
    A[i] = int(A[i])
```

Используя функции языка **map** и **list** то же самое можно сделать в одну строку:

```
A = list(map(int, input().split()))
```

У метода split есть необязательный параметр, который определяет, какая строка будет использоваться в качестве разделителя между элементами списка.

Например, метод split('.') вернет список, полученный разрезанием исходной строки по символам '.'

Отметим, что «разбивать тестовую строку на слова» этим методом не удобно, т.к. в качестве разделителей обычно применяется не один символ.

Используя “обратные” методы можно вывести список при помощи однострочной команды.

Для этого используется метод строки **join**. У этого метода один параметр: список строк. В результате получается строка, полученная соединением элементов списка (которые переданы в качестве параметра) в одну строку, при этом между элементами списка вставляется разделитель, равный той строке, к которой применяется метод. Например программа

```
A = ['red', 'green', 'blue']
print('  '.join(A))
print(''.join(A))
print('***'.join(A))
```

выведет строки

```
red green blue
redgreenblue
red***green***blue
```

Если же список состоит из чисел, то придется использовать еще и функцию **map**. Вывести элементы списка чисел, разделяя их пробелами, можно так:

```
print(   ' '.join(  map(str, A)  )  )
```

Генераторы списков

Для создания списка, заполненного одинаковыми элементами, можно использовать оператор повторения списка:

A **=** [0] * n

Для создания списков, заполненных по более сложным формулам можно использовать **генераторы**: выражения, позволяющие заполнить список некоторой формулой.

Общий вид генератора следующий:

[*выражение* for *переменная* in *список*]

где переменная — идентификатор некоторой переменной,

список — список значений, который принимает данная переменная (как правило, полученный при помощи функции **range**),

выражение — некоторое выражение, которым будут заполнены элементы списка, как правило, зависящее от использованной в генераторе переменной.

Примеры генераторов списка

Создать список, состоящий из n нулей:

```
A = [ 0 for i in range(n)]
```

Создать список, заполненный квадратами целых чисел:

```
A = [ i ** 2 for i in range(n)]
```

Если нужно заполнить список квадратами чисел от 1 до n, то можно изменить параметры функции **range** на **range**(1, n + 1):

```
A = [ i ** 2 for i in range(1, n + 1)]
```

Список, заполненный случайными числами от 1 до 9 (используя функцию randint из модуля random):

```
A = [ randint(1, 9) for i in range(n)]
```

Создать список из строк, считанных со стандартного ввода: сначала нужно ввести число элементов списка (это значение будет использовано в качестве аргумента функции **range**), потом — заданное количество строк:

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

Срезы

Со списками можно делать срезы.

А именно:

A[i:j] срез из j-i элементов A[i], A[i+1], ..., A[j-1].

A[i:j:-1] срез из i-j элементов A[i], A[i-1], ..., A[j+1] (то есть меняется порядок элементов).

A[i:j:k] срез с шагом k: A[i], A[i+k], A[i+2*k],... . Если значение k<0, то элементы идут в противоположном порядке.

Каждое из чисел i или j может отсутствовать, что означает “начало строки” или “конец строки”

Списки, в отличии от строк, являются изменяемыми объектами: можно отдельному элементу списка присвоить новое значение.

Но можно менять и целиком срезы. Например:

A = [1, 2, 3, 4, 5]

A[2:4] = [7, 8, 9]

Получится список, у которого вместо двух элементов среза A[2:4] вставлен новый список уже из трех элементов.

Теперь список стал равен [1, 2, 7, 8, 9, 5].

A = [1, 2, 3, 4, 5, 6, 7]

A[::-2] = [10, 20, 30, 40]

Получится список [40, 2, 30, 4, 20, 6, 10].

Здесь A[::-2] — это список из элементов A[-1], A[-3], A[-5], A[-7], которым присваиваются значения 10, 20, 30, 40 соответственно.

Если не непрерывному срезу (то есть срезу с шагом k, отличному от 1), присвоить новое значение, то количество элементов в старом и новом срезе обязательно должно совпадать, в противном случае произойдет ошибка ValueError.

Обратите внимание, A[i] — это элемент списка, а не срез!

Основные операции со списками

`x in A`	Проверить, содержится ли элемент в списке. Возвращает True или False
`x not in A`	То же самое, что not(x in A)
`min(A)`	Наименьший элемент списка
`max(A)`	Наибольший элемент списка
`A.index(x)`	Индекс первого вхождения элемента x в список, при его отсутствии генерирует исключение `ValueError`
`A.count(x)`	Количество вхождений элемента x в список

Обработка и вывод вложенных списков

Часто в задачах приходится хранить прямоугольные таблицы с данными

Такие таблицы называются матрицами или двумерными массивами.

В языке программирования Питон таблицу можно представить в виде списка строк, каждый элемент которого является в свою очередь списком

Например, создать числовую таблицу из двух строк и трех столбцов можно так:

A **=** [[1, 2, 3], [4, 5, 6]]

Здесь первая строка списка A[0] является списком из чисел [1, 2, 3]:

A[0][0] == 1 A[0][1] == 2 A[0][2] == 3

A[1][0] == 4, A[1][1] == 5 A[1][2] == 6.

Для обработки и вывода списка как правило используется два вложенных цикла.

Первый цикл по номеру строки, второй цикл по элементам внутри строки.

Например, вывести двумерный числовой список на экран построчно, разделяя числа пробелами внутри одной строки:

```
for i in range(len(A)):
    for j in range(len(A[i]):
        print(A[i][j], end = ' ')
    print()
```

То же самое, но циклы не по индексу, а по значениям списка:

```
for row in A:
    for elem in row:
        print(elem, end = ' ')
    print()
```

Естественно для вывода одной строки можно воспользоваться методом join:

```
for row in A:
    print(' '.join(list(map(str, row))))
```

Используем два вложенных цикла для подсчета суммы всех чисел в списке:

```
S = 0
for i in range(len(A)):
    for j in range(len(A[i])):
        S += A[i][j]
```

Или то же самое с циклом не по индексу, а по значениям строк:

```
S = 0
for row in A:
    for elem in row:
        S += elem
```

Создание вложенных списков

Пусть даны два числа: количество строк n и количество столбцов m. Необходимо создать список размером n×m, заполненный нулями.

Очевидное **решение оказывается неверным**:

A **=** [[0] * m] * n

В этом легко убедиться, если присвоить элементу A[0][0] значение 1, а потом вывести значение другого элемента A[1][0] — оно тоже будет равно 1

Дело в том, что **[0] * m *возвращает ссылку на список из m нулей***.

Но последующее повторение этого элемента создает список из n элементов, которые являются ссылкой на один и тот же список

Точно так же, как выполнение операции B **=** A для списков не создает новый список

Поэтому все строки результирующего списка на самом деле являются одной и той же строкой.

Таким образом, двумерный список нельзя создавать при помощи операции повторения одной строки.

Первый способ: сначала создадим список из n элементов (для начала просто из n нулей).

Затем сделаем каждый элемент списка ссылкой на другой одномерный список из m элементов:

```
A = [0] * n
for i in range(n):
    A[i] = [0] * m
```

Другой (но похожий) способ: создать пустой список, потом n раз добавить в него новый элемент, являющийся списком-строкой:

```
A = []
for i in range(n):
    A.append([0] * m)
```

Можно воспользоваться генератором: создать список из n элементов, каждый из которых будет списком, состоящих из m нулей:

```
A = [ [0] * m for i in range(n)]
```

В этом случае каждый элемент создается независимо от остальных (заново конструируется список **`[0] * m`** для заполнения очередного элемента списка), а не копируются ссылки на один и тот же список.

Ввод двумерного списка (массива)

Пусть программа получает на вход двумерный массив, в виде n строк, каждая из которых содержит m чисел, разделенных пробелами. Как их считать?

```
A = []
for i in range(n):
    A.append(list(map(int, input().split())))
```

Или, без использования сложных вложенных вызовов функций:

```
A = []
for i in range(n):
    row = input().split()
    for i in range(len(row)):
        row[i] = int(row[i])
    A.append(row)
```

Можно сделать то же самое и при помощи генератора:

```
A = [ list(map(int, input().split())) for i in range(n)]
```

Пример обработки двумерного массива

Пусть дан квадратный массив из n строк и n столбцов. Необходимо элементам, находящимся на главной диагонали, проходящей из левого верхнего угла в правый нижний (то есть тем элементам A[i][j], для которых i==j) присвоить значение 1, элементам, находящимся выше главной диагонали – значение 0, элементам, находящимся ниже главной диагонали – значение 2. То есть получить такой массив (пример для n==4):

1 0 0 0
2 1 0 0
2 2 1 0
2 2 2 1

Элементы, которые лежат выше главной диагонали – это элементы A[i][j], для которых i<j, а для элементов ниже главной диагонали i>j. Таким образом, мы можем сравнивать значения i и j и по ним определять значение A[i][j]. Получаем следующий алгоритм:

```
for i in range(n):
    for j in range(n):
        if i < j:
            A[i][j] = 0
        elif i > j:
            A[i][j] = 2
        else:
            A[i][j] = 1
```

Данный алгоритм плох, поскольку выполняет одну или две инструкции **if** для обработки каждого элемента. Если мы усложним алгоритм, то мы сможем обойтись вообще без условных инструкций.

Сначала заполним главную диагональ, для чего нам понадобится один цикл:

```
for i in range(n):
    A[i][i] = 1
```

Затем заполним значением 0 все элементы выше главной диагонали, 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
```

Можно также внешние циклы объединить в один и получить еще одно, более компактное решение:

```
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
```

Это решение использует операцию повторения списков для построения очередной строки списка.

i-я строка списка состоит из i чисел 2,

затем идет одно число 1,

затем идет n-i-1 число 0:

```
for i in range(n):
    A[i] = [2] * i + [1] + [0] * (n - i - 1)
```

А можно заменить цикл на генератор:

```
A = [ [2] * i + [1] + [0] * (n - i - 1)                    \
                                          for i in  range(n) ]
```

Вложенные генераторы двумерных массивов

Для создания двумерных массивов можно использовать вложенные генераторы, разместив генератор списка, являющегося строкой, внутри генератора для строк.

Например, сделать список из n строк и m столбцов при помощи генератора, создающего список из n элементов, каждый элемент которого является списком из m нулей:

```
[ [0] * m for i in range(n)]
```

Но при этом внутренний список также можно создать при помощи, например, такого генератора:

[0 **for** j **in range**(m)].

Вложив один генератор в другой получим вложенные генераторы:

[[0 **for** j **in** range(m)] **for** i **in** range(n)]

Но если число 0 заменить на некоторое выражение, зависящее от i (номер строки) и j (номер столбца), то можно получить список, заполненный по некоторой формуле.

Например, пусть нужно задать следующий массив (для удобства добавлены дополнительные пробелы между элементами):

```
0   0   0   0   0   0
0   1   2   3   4   5
0   2   4   6   8  10
0   3   6   9  12  15
0   4   8  12  16  20
```

В этом массиве n = 5 строк, m = 6 столбцов, и элемент в строке i и столбце j вычисляется по формуле: A[i][j] = i * j.

Для создания такого массива можно использовать генератор:

```
[[ i * j for j in range(m)]      \
                  for i in range(n)]
```

Файловый ввод-вывод

-