Python

Язык программирования Python был создан в 1991 году голландцем Гвидо ван Россумом.
Пайтон (или Питон) получил название от комедийных серий BBC "Летающий цирк Монти-Пайтона ". Python активно совершенствуется и в настоящее время. Официальный сайт **http://python.org**.

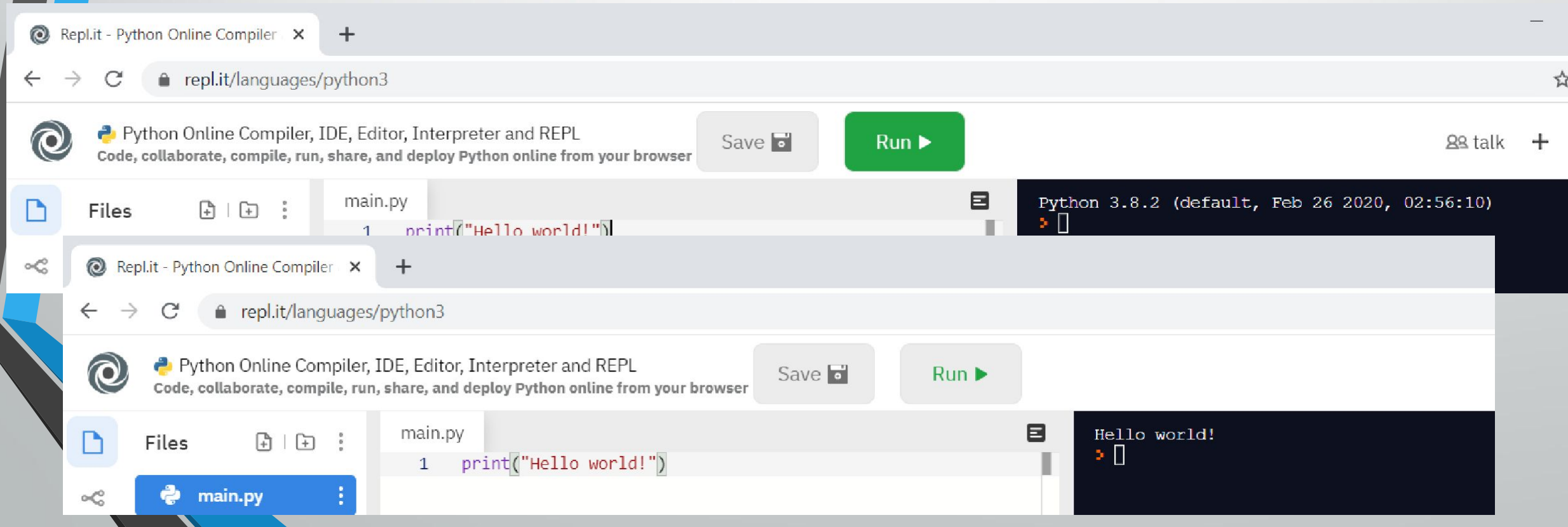
Про Python

Интерпретатор выполняет команды построчно: пишешь строку, нажимаешь *Enter*, интерпретатор выполняет ее, наблюдаешь результат. Python можно использовать как калькулятор.

Другой вариант работы в интерактивном режиме — это работа в среде разработки IDLE (Integrated Development and Learning Environment).

Запускаем IDLE после чего уже можно начинать писать первую программу.

Традиционно, первой программой будет “hello world”. Чтобы написать“hello world” на python, достаточно всего одной строки: print("Hello world!").



Правила записи инструкций

- Конец строки является концом инструкции.
- Вложенные инструкции объединяются в блоки по величине отступов.
- Отступ может быть любым, главное, чтобы в пределах одного вложенного блока отступ был одинаков. Отступ в один пробел - не лучшее решение. Используйте четыре пробела (знак табуляции).
- Вложенные инструкции в Python записываются в соответствии с одним и тем же шаблоном, когда основная инструкция завершается двоеточием, вслед за которым располагается вложенный блок кода, обычно с отступом под строкой основной инструкции.
- Можно записать несколько инструкций в одной строке, разделяя их точкой с запятой:
  a = 1; b = 2; print(a, b)
- Допустимо записывать одну инструкцию в нескольких строках. Достаточно ее заключить в пару круглых, квадратных или фигурных скобок:

```
if (a == 1 and b == 2 and
c == 3 and d == 4): # Не забываем про двоеточие
    print('spam' * 3)
```

- Тело со ставной инструкции может располагаться в той же строке, что и тело основной, если тело составной инструкции не содержит со ставных инструкций:
  if x > y: print(x)

Данные

Представлены константами и переменными.
Константы
целые числа: 4   687 -45 0
числа с плавающей точкой:
с фиксированной точкой 1.45, -3.789654, 0.00453
с плавающей точкой 1.0E-5, -5.123e2, 0.1234E3
строки: "red", "What is your name?", ‘Привет!', '1234‘ (кавычки в Python могут быть одинарными или двойными)
логические: True, False
списки: [1, -34, 22], [0, 0], []
кортежи: (1, -34, 22), (0,0), ()

Все данные в Python представляют собой объекты. Каждый объект содержит как минимум три вида данных:

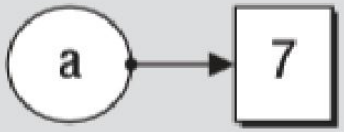
- счётчик ссылок;
- тип;
- значение.

Счётчик ссылок используется для управления памятью.

Данные

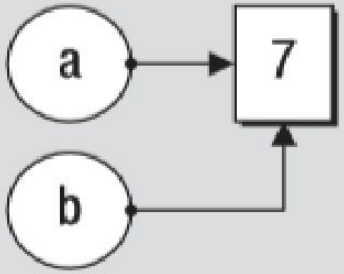
Переменные. В языке Python нет переменных как таковых – вместо них используются *ссылки на объекты*. Когда речь заходит о неизменяемых объектах, таких как `int` или `str`, между переменной и ссылкой на объект нет никакой разницы. Однако различия начинают проявляться, когда дело доходит до изменяемых объектов, но эти различия редко имеют практическое значение. Мы будем использовать термины *переменная* и *ссылка на объект* как взаимозаменяемые.

```
a = 7
```

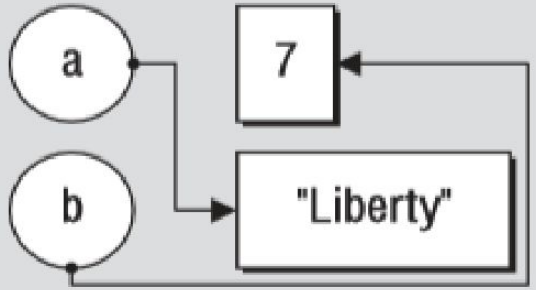


Кружочками изображены ссылки на объекты.
Прямоугольниками – объекты в памяти.

```
a = 7
b = a
```




```
a = 7
b = a
a = "Liberty"
```



Имена

Имя может начинаться с латинской буквы (любого регистра) или подчеркивания, а дальше допустимо использование цифр. В качестве идентификаторов нельзя применять ключевые слова языка и нежелательно переопределять встроенные имена. Список ключевых слов:

```
>>> import keyword
>>> keyword.kwlist
['and', 'assert', 'break', 'class', 'continue', 'def', 'del',
'elif', 'else', 'except', 'exec', 'finally', 'for', 'from',
'global', 'if', 'import', 'in', 'is', 'lambda', 'not', 'or',
'pass', 'print', 'raise', 'return', 'try', 'while', 'yield']
```

Имена, начинающиеся с подчеркивания или двух подчеркиваний, имеют особый смысл. Одиночное подчеркивание говорит программисту о том, что имя не должно использоваться за пределами модуля.

Имена

Переменную можно связать с объектом в любом месте блока, важно, чтобы это произошло до ее использования, иначе будет возбуждено исключение NameError. Связывание имен со значениями происходит в инструкциях присваивания.

Правила:

- Всегда следует связывать переменную со значением до ее использования.
- Необходимо избегать глобальных переменных и передавать все в качестве параметров. Глобальными на уровне модуля должны остаться только имена-константы, имена классов и функций.
- Убрать связь имени с объектом можно с помощью оператора del. В этом случае, если объект не имеет других ссылок на него, он будет удален. Для управления памятью в Python используется **подсчет ссылок** (*reference counting*), для удаления наборов объектов с зацикленными ссылками - **сборка мусора** (*garbage collection*).

Имена

В каждой точке программы интерпретатор "видит" три **пространства имен**: локальное, глобальное и встроенное. Пространство имен связано с понятием **блока кода**. В Python блоком кода является то, что исполняется как единое целое, например, тело определения функции, класса или модуля.

Локальные имена - имена, которым присвоено значение в данном блоке кода. *Глобальные имена* - имена, определяемые на уровне блока модуля или те, которые явно заданы в операторе ***global***. Встроенные имена - имена из специального словаря ***__ builtins __***.

Области видимости имен могут быть вложенными друг в друга, например, внутри вызванной функции видны имена, определенные в вызывающем коде. Переменные, которые используются в блоке кода, но связаны со значением вне кода, называются **свободными переменными**.

Арифметические типы: int, long, float. Логический тип: bool
Неизменяемые типы

Пример 1:

```
a = str(10)
print (type(a))
b=-11
print ("1: a=",id(a)," ",a," b=",id(b)," ",b)
a=b
print (type(a))
print ("2: a=",id(a)," ",a," b=",id(b)," ",b)
a=22
print (type(a))
print ("3: a=",id(a)," ",a," b=",id(b)," ",b)
```

```
<class 'str'>
1: a= 139730666731120   10  b= 139730667379216   -11
<class 'int'>
2: a= 139730667379216   -11  b= 139730667379216   -11
<class 'int'>
3: a= 139731396614528   22  b= 139730667379216   -11
```

```
r= True   v= False
r= False   v= False
r= False   z= False
r= False   z= True
```

Строки: str (Unicode UTF-8, UTF-16, UTF-32). Неизменяемый тип

```
x="красный"
y="зеленый"
z="синий"
print ("x: ",id(x)," ", x," y: ",id(y)," ", y," z: ",id(z)," ", z)
y=x
print ("x: ",id(x)," ", x," y: ",id(y)," ", y," z: ",id(z)," ", z)
z=y
print ("x: ",id(x)," ", x," y: ",id(y)," ", y," z: ",id(z)," ", z)
z="синий"
print ("x: ",id(x)," ", x," y: ",id(y)," ", y," z: ",id(z)," ", z)
z="12346"
print ("x: ",id(x)," ", x," y: ",id(y)," ", y," z: ",id(z)," ", z)
```

```
x:  139873442909408  красный  y:  139873441999648  зеленый  z:  139873442084880  синий
x:  139873442909408  красный  y:  139873442909408  красный  z:  139873442084880  синий
x:  139873442909408  красный  y:  139873442909408  красный  z:  139873442909408  красный
x:  139873442909408  красный  y:  139873442909408  красный  z:  139873442084880  синий
x:  139873442909408  красный  y:  139873442909408  красный  z:  139873442170672  12346
```

Списки: list. Кортежи: tulple. Изменяемые типы

```
a=b=[1, -2]
print (" a=",a," b= ", b)
b[0]=-99
print (" a=",a," b= ", b)
c=[123, -987]
print (" a=",a," b= ", b, " c= ", c)
c=a
print (" a=",a," b= ", b, " c= ", c)
```

```
a= [1, -2]  b=  [1, -2]
a= [-99, -2]  b=  [-99, -2]
a= [-99, -2]  b=  [-99, -2]  c=  [123, -987]
a= [-99, -2]  b=  [-99, -2]  c=  [-99, -2]
```

```
a= (1, -2)  b=  (1, -2)
a= (1, -2)  b=  (-99, 77)
a= (1, -2)  b=  (-99, 77)  c=  (123, -987)
a= (1, -2)  b=  (-99, 77)  c=  (1, -2)
```

```
c=(123, -987)
print (" a=",a," b= ", b, " c= ", c)
c=a
```

Пример:

a = 2; b = a; b = 3

print ("семантика копирования: a=",a,' b=',b)

a = [2,5]; b = a; b[0] = 3

print ("семантика указателей: a=",a,' b=',b)

```
семантика копирования: a= 2     b= 3
семантика указателей: a= [3, 5]     b= [3, 5]
```

Целые числа (int)

Целые числа поддерживают длинную арифметику (это требует больше памяти)

x + y	Сложение
x - y	Вычитание
x * y	Умножение
x / y	Деление
x // y	Получение целой части от деления
x % y	Остаток от деления
-x	Смена знака числа
abs(x)	Модуль числа
divmod(x, y)	Пара (x // y, x % y)
x ** y	Возведение в степень
pow(x, y[, z])	x^y по модулю (если модуль задан)

Целые числа (int)

Над целыми числами можно производить битовые операции:

x | y	Побитовое *или*
x ^ y	Побитовое *исключающее или*
x & y	Побитовое *и*
x << n	Битовый сдвиг влево
x >> y	Битовый сдвиг вправо
~x	Инверсия битов

Пример

```
x=7
y=3
z=x/y
print ("x= ", x," y= ", y," x/y= ", z)
z=x//y
print ("x= ", x," y= ", y," x//y= ", z)
z=x%y
print ("x= ", x," y= ", y," x%y= ", z)
z=x**y
print ("x= ", x," y= ", y," x**y= ", z)
z=pow(x,y,2)
print ("x= ", x," y= ", y," pow(x,y,2)= ", z)
```

```
x=  7  y=  3  x/y=  2.3333333333333335
x=  7  y=  3  x//y=  2
x=  7  y=  3  x%y=  1
x=  7  y=  3  x**y=  343
x=  7  y=  3  pow(x,y,2)=  1
```

Вещественные числа (float)

```
import math
x=7.02
y=3.5
z=x/y
print ("x= ", x," y= ", y," x/y= ", z)
z=math.pi
print ("pi= ", z)
z=math.sqrt(x*y)
print ("x= ", x," y= ", y," sqrt(x*y)= ", z)
z=pow(x,y)
print ("x= ", x," y= ", y," pow(x,y)= ", z)
z=0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1
print ("z= ", z)
```

```
x=  7.02  y=  3.5  x/y=  2.005714285714286
pi=  3.141592653589793
x=  7.02  y=  3.5  sqrt(x*y)=  4.956813492557492
x=  7.02  y=  3.5  pow(x,y)=  916.6000834628456
z=  0.9999999999999999
```

Логические значения (bool)

Операции сравнения:

> (больше), < (меньше), >= (больше или равно), <= (меньше или равно), == (равно), ***!***= (не равно)

Логические операции:

- конкатенация

  X ***and*** Y

Истина, если оба значения X иY истинны;

- дизъюнкция

  X ***or*** Y

Истина, если хотя бы одно из значений X или Y истинно;

- инверсия (отрицание)

  not X

Истина, если X ложно.

Комплексные числа (complex)

```
import cmath
x=7.02
y=3.5
z = complex(x, y)
print(z)
z = x + y
print(z)
z = x * y
print(z)
z = x / y
print(z)
z = complex(x, y)
print(z.conjugate()) # Сопряжённое число
print(z.imag)
print(z.real)
```

```
(7.02+3.5j)
10.52
24.57
2.005714285714286
(7.02-3.5j)
3.5
7.02
```

Строки. Функции и методы строк (str)

- Конкатенация (сложение)

```
s="Python+"
st="Пайтон"
print(s+st)                #Python+ Пайтон
```

- Дублирование строки

```
st="Пайтон"
print(st*3)                # Пайтон Пайтон Пайтон
```

- Длина строки

```
st="Пайтон"
print(len(st))             #6
```

- Доступ по индексу

```
s="Python+"
st="Пайтон"
print(s[0]," ",st[3]," ",st[-4])   #P т й
```

Строки. Функции и методы строк (str)

- Извлечение среза

Оператор извлечения среза: [X:Y] X–начало среза, а Y–окончание;

```
s="Python+"
st="Пайтон"
print(s[3:5]," ",st[2:-2]," ",s[-1])   #ho йт +
```

По умолчанию первый индекс равен 0, а второй –длине строки

```
s="Python+"
st="Пайтон"
print(s[:5]," ",st[1:]," ",s[:])          #Pytho айтон Python+
```

Кроме того, можно задать шаг, с которым нужно извлекать срез

```
s="Python+"
print(s[2::2])                    #to+
```

Строки. Форматирование строк с помощью метода format

- Форматирование строк с помощью метода format

Список вывода

```
print ('{0}, {1}, {2}'.format('a', 'b', 'c'))
print ('{}, {}, {}'.format('a', 'b', 'c'))
print ('{2}, {1}, {0}'.format('a', 'b', 'c'))
print ('{0}{1}{0}'.format('1234', 'Good'))
```

```
a, b, c
a, b, c
c, b, a
1234Good1234
```

Параметры метода format:

поле замены ::= "{" [имя поля] ["!" преобразование] [":" спецификация] "}"
выравнивание ::= "<" | ">" | "=" | "^" знак ::= "+" | "-" | " "
тип ::= "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" | "n" | "o" | "s" | "x" | "X" | "%"

Строки

Выравнивание производится при помощи символа заполнителя

Флаг	Значение
‘<’	Символы-заполнители будут справа (выравнивание объекта по левому краю) (по умолчанию).
‘>’	выравнивание объекта по правому краю.
‘=’	Заполнитель будет после знака, но перед цифрами. Работает только с числовыми типами.
‘^’	Выравнивание по центру.

Опция “знак” используется только для чисел

Флаг	Значение
‘+’	Знак должен быть использован для всех чисел.
‘-‘	‘-‘ для отрицательных, ничего для положительных.
‘Пробел’	‘-‘ для отрицательных, пробел для положительных.

Строки

Поле “тип” может принимать следующие значения

Тип	Значение
‘d’, ‘i’, ‘u’	Десятичное число.
‘o’	Число в восьмеричной системе счисления.
‘x’	Число в шестнадцатеричной системе счисления (буквы в нижнем регистре).
‘X’	Число в шестнадцатеричной системе счисления (буквы в верхнем регистре).
‘e’	Число с плавающей точкой с экспонентой (экспонента в нижнем регистре).
‘E’	Число с плавающей точкой с экспонентой (экспонента в верхнем регистре).
‘f’, ‘F’	Число с плавающей точкой (обычный формат).
‘g’	Число с плавающей точкой. с экспонентой (экспонента в нижнем регистре), если она меньше, чем -4 или точности, иначе обычный формат.
‘G’	Число с плавающей точкой. с экспонентой (экспонента в верхнем регистре), если она меньше, чем -4 или точности, иначе обычный формат.
‘c’	Символ (строка из одного символа или число - код символа).
‘s’	Строка.
‘%’	Число умножается на 100, отображается число с плавающей точкой, а за ним знак %.

Пример

```
print('{:<30}'.format('left aligned'))
print('{:>30}'.format('right aligned'))
print('{:^30}'.format('centered'))
print('{:*^30}'.format('centered'))
print('{:+f}; {:+f}'.format(3.14, -3.14))
print('{: f}; {: f}'.format(3.14, -3.14))
print('{:-f}; {:-f}'.format(3.14, -3.14))
print("int: {0:d}; hex: {0:#x}; oct: {0:#o}; bin: {0:#b}".format(42))
points = 19.5
total = 22
print('Correct answers: {:.2%}'.format(points/total))
```

```
left aligned
                  right aligned
             centered
***********centered***********
+3.140000; -3.140000
 3.140000; -3.140000
3.140000; -3.140000
int: 42; hex: 0x2a; oct: 0o52; bin: 0b101010
Correct answers: 88.64%
```

Ввод данных с клавиатуры. Функция input

Ввод данных с клавиатуры в программу (начиная с версии Python 3.0) осуществляется с помощью функции ***input()***. Когда данная функция выполняется, то поток выполнения программы останавливается в ожидании данных, которые пользователь должен ввести с помощью клавиатуры. После ввода данных и нажатия <Enter>, функция input() завершает свое выполнение и возвращает результат, который представляет собой **строку символов**, введенных пользователем. Если требуется получить число, то результат выполнения функции input() изменяют с помощью функций ***int()*** или ***float()***.

Результат, возвращаемый функцией input(), обычно присваивают переменной для дальнейшего использования в программе.

Когда выполняющаяся программа предлагает пользователю что-либо ввести, то пользователь может не понять, что от него хотят. Надо как-то сообщить, ввод каких данных ожидает программа. С этой целью функция input() может принимать необязательный аргумент-приглашение строкового типа; при выполнении функции сообщение будет появляться на экране и информировать человека о запрашиваемых данных.

Ввод данных с клавиатуры. Функция input

Пример

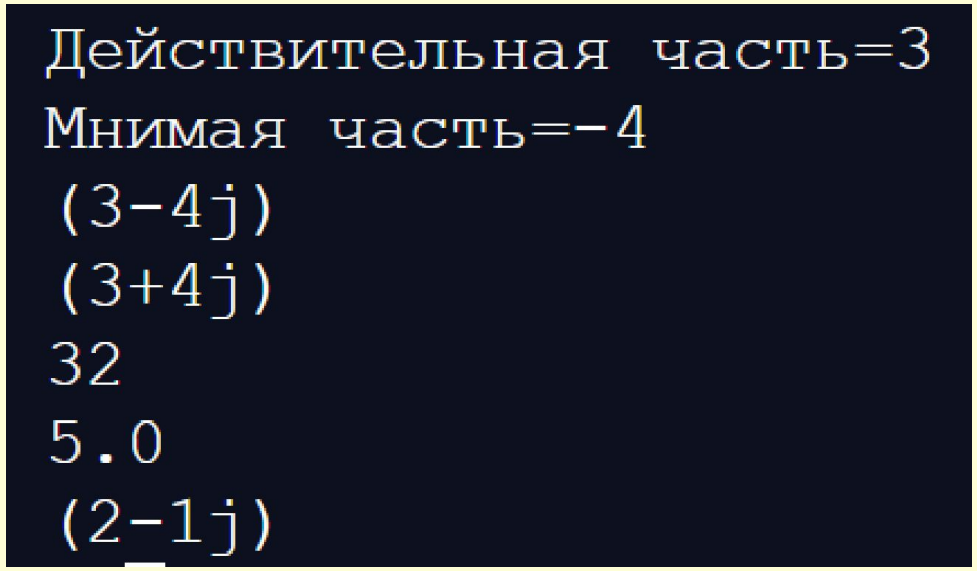
```
st=input()
print("st=", st, "тип: ",type(st))
a=int(input("a="))
print("a=", a, "тип: ",type(a))
b=float(input("b="))
print("b=", b, "тип: ",type(b))
z=a+b
print ("a+b=",z)
z=st+a
print ("st+a=",z)
```

```
-123
st= -123 тип:  <class 'str'>
a=-57
a= -57 тип:  <class 'int'>
b=234.56
b= 234.56 тип:  <class 'float'>
a+b= 177.56
Traceback (most recent call last):
  File "main.py", line 9, in <module>
    z=st+a
TypeError: can only concatenate str (no
t "int") to str
```

Ввод данных с клавиатуры. Функция input

Пример

```
import cmath
import math
x=float(input("Действительная часть="))
y=float(input("Мнимая часть="))
z = complex(x, y)
print(z)
print(z.conjugate())
print(z.__sizeof__()) #занимаемая память
print(math.sqrt(x*x+y*y))
print(cmath.sqrt(z))
```



Операции. Присваивание. Выражение

Операция - это выполнение каких-нибудь действий над данными (операндами). Для выполнения конкретных действий требуются специальные инструменты -операторы. В программе на языке Python связь между данными и переменными задается с помощью знака =. Такая операция называется присваиванием.

Примеры:

b=-11; a=b; a = str(10)

a=b=[1, -2]

Выражение – это формула для вычисления значения. Она образуется из операндов, соединенных знаками операций и выражений в круглых скобках. В качестве операндов могут выступать переменные, константы, указатели функций.

Тип переменной в левой части оператора присваивания обычно должен совпадать с типом значения выражения в правой части. Возможны случаи несовпадения типов, например, когда слева переменная вещественного типа, а справа выражение целого типа. Выражения являются составной частью операторов. Вычисление выражений осуществляется слева направо, за исключением операции возведения в степень (справа налево), с учетом наличия круглых скобок и приоритетом операций.

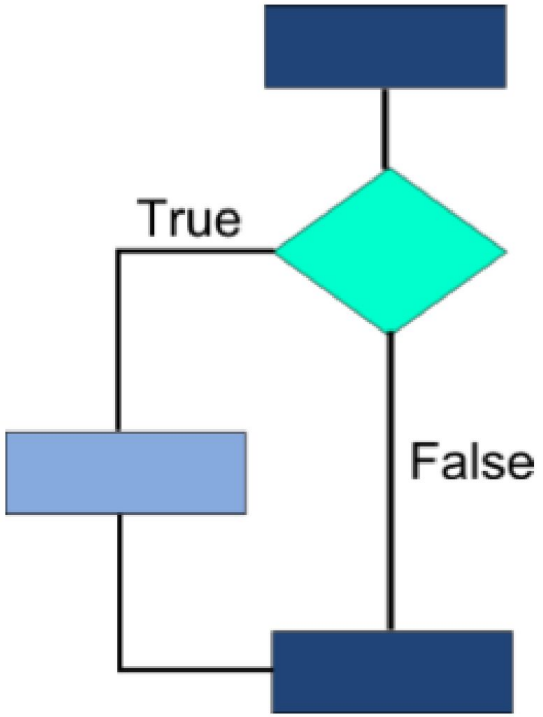
Операции. Приоритет операций

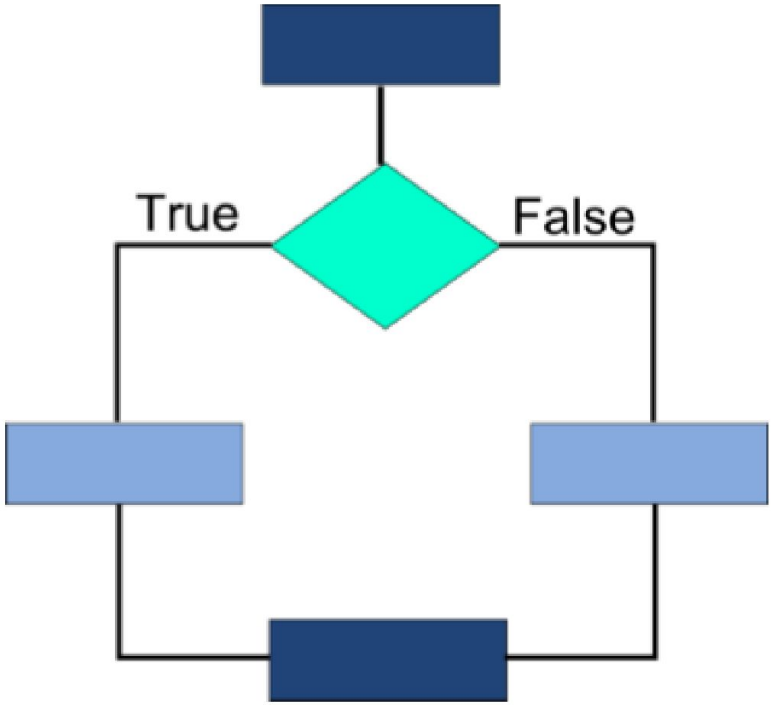
Операция	Название	Приоритет
lambda	лямбда-выражение	низкий
or	логическое ИЛИ	
and	логическое И	
not x	логическое НЕ	
in, not in	проверка принадлежности	
is, is not	проверка идентичности	
<, <=, >, >=, !=, ==	сравнения	
|	побитовое ИЛИ	
^	побитовое исключающее ИЛИ	
&	побитовое И	
<<, >>	побитовые сдвиги	
+, -	сложение и вычитание	
*, /, %, //	умножение, деление, остаток	
+x, -x	унарный плюс и смена знака	
~x	побитовое НЕ	
**	возведение в степень	высокий

Условный оператор. Инструкция if … else

```
if условие :
    оператор
```

```
if условие :
    оператор
else:
    оператор
```





Условный оператор. Инструкция if … else

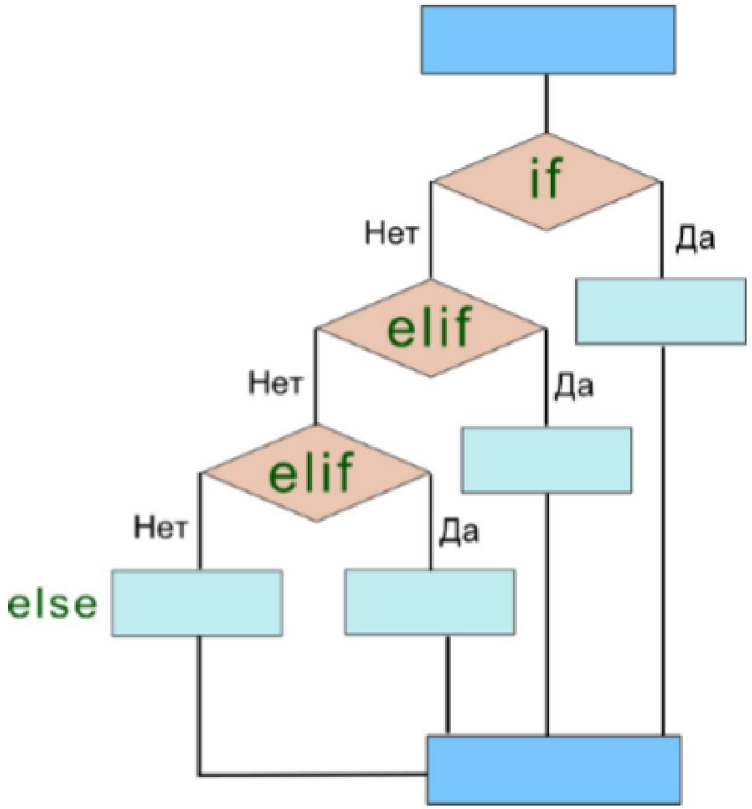
Пример. Найти наибольшее из двух значений

```
a=1; b= 23
if a>b:
    z=a
else:
    z=b
print ("max1=",z)
a=-123; b=-5
z=a if (a>b) else b
print ("max2=",z)
```

```
max1= 23
max2= -5
```

Множественное ветвление

Данная расширенная инструкция, помимо необязательной части else, содержит ряд ветвей elif (сокращение от "else if" - "еще если").
В отличии от использования множества одиночных инструкций if, инструкция if … elif … else прекращает просмотр последующих ветвей, как только логическое выражение в текущей ветке вернет True.



Множественное ветвление

Пример. Простейший калькулятор

```
result = "Нет такой операции"
n = 3; m=-12
op=":" #  op="**"  op="^"
if op == "+": result = n+m
elif op == "-": result = n-m
elif op == "*": result = n*m
elif op == ":": result = n/m
elif op == "div": result = n//m
elif op == "^": result = m**n
else: print ("Error")
print("Результат=", result)
```

```
Результат= -0.25
```

```
Error
Результат= Нет такой операции
```

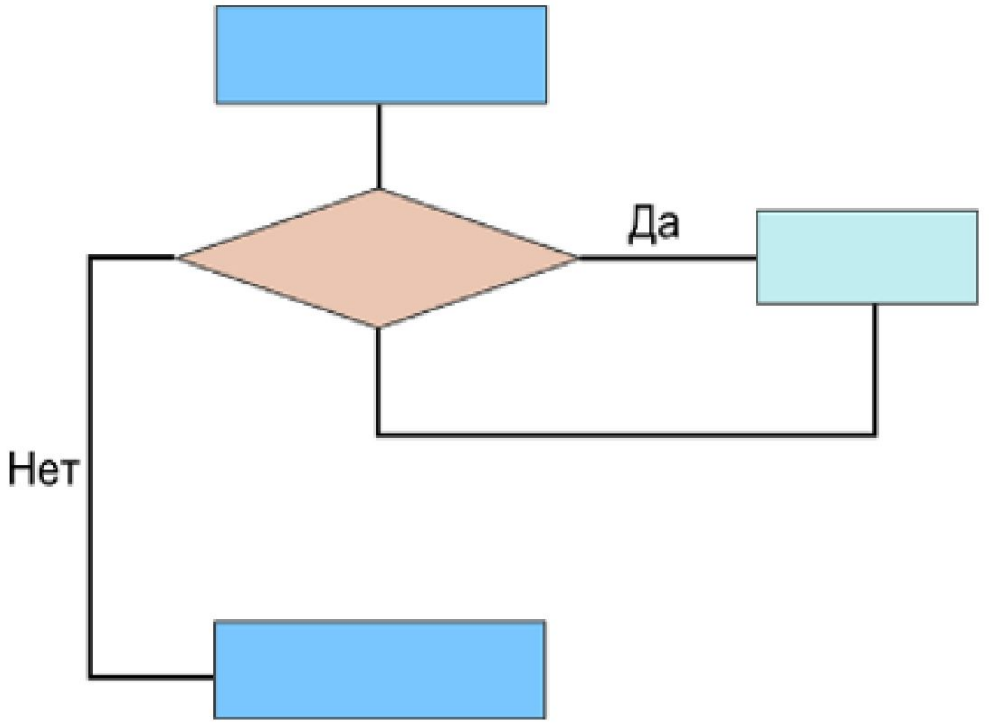
```
Результат= -1728
```

Цикл while

While-один из самых универсальных циклов в Python, поэтому довольно медленный. Выполняет тело цикла до тех пор, пока условие цикла истинно.

```
s = 0
r=0
i = 0
while i < 10:
    s+=3
    r+=s
    print (s)
    i+=2
print ("i=", i," r=",r)
```

```
3
6
9
12
15
i= 10    r= 45
```



Цикл for. Функция range()

Цикл for сложнее, но менее универсальный, выполняется гораздо быстрее цикла while. Часто в цикле for используется функция range().

Есть три способа вызова range():

- range(***конец***) один аргумент, начало = 0

```
for i in range(3):
    print(i, end='  ')
```

```
0  1  2
```

- range(***начало***, ***конец***) два аргумента

```
for i in range(1,4):
 print(i, end='  ')
```

```
1  2  3
```

- range(***начало***, ***конец***, ***шаг***) три аргумента

```
for i in range(3, 16, 3):
    q = i / 4
    print(f"{i} делится на 3, результат {int(q)}.")
```

```
3 делится на 4 результат 0.
6 делится на 4 результат 1.
9 делится на 4 результат 2.
```

Цикл for

Этот цикл проходится по любому итерируемому объекту (например, строке или списку),и во время каждого прохода выполняет тело цикла.

```
stroka = "привет"
 for b in stroka:
   print(b, end=' * ')
```

```
п * р * и * в * е * т *
```

Найти сумму цифр числа, заданного строкой знаков.

```
ta=0
for i in "654321":
ta+=int(i)
print(ta)
```

```
21
```

Оператор continue. Оператор break

Оператор ***continue*** начинает следующий проход цикла, минуя оставшееся тело цикла for или while.

Оператор ***break*** досрочно прерывает цикл.

Слово else, примененное в цикле for или while, проверяет, был ли произведен выход из цикла инструкцией break, или же “естественным” образом. Блок инструкций внутри else выполнится только в том случае, если выход из цикла произошел без помощи break.

Пример

```
for i in 'hello world':
if i == 'a':
    print('Буква a в строке есть')
    break
  else:
      print('Это не буква a ')
      continue
```

```
Это не буква a
Это не буква a
Это не буква a
Это не буква a
Это не буква a
Это не буква a
Это не буква a
Это не буква a
Это не буква a
Это не буква a
Это не буква a
>
```

'hallo world':

```
Это не буква a
Буква a в строке есть
>
```

Функции. Инструкция def

Функции в программировании можно представить как изолированный блок кода, обращение к которому в процессе выполнения программы может быть многократным.

Описание функции

Программист может определять собственные функции двумя способами: с помощью оператора ***def*** или прямо в выражении, посредством ***lambda***.

def – это инструкция (команда) языка программирования Python, позволяющая создавать функцию.

Имя функции может быть любым, но желательно осмысленным.

После имени в круглых скобках перечисляются параметры функции. Если их нет, то скобки остаются пустыми.

Далее идет двоеточие, обозначающее окончание заголовка функции (аналогично с условиями и циклами).

После заголовка с новой строки и с отступом следуют инструкции тела функции.

В функции присутствует инструкция ***return*** (может и не быть), которая возвращает значение в основную ветку программы.

Вызов функции

Состоит из имени функции и списка фактических параметров, заключенного в круглые скобки.

Функции

```
def max(x, y):
  z=x if x>y else y
  return z

def newfunc(n):
  def myfunc(x):
    return x + n
  return myfunc

print(max(1, 10))
print(max("Xa-xa","Ky-Ky" ))
n = newfunc(500)
print(n)
print(n(-111))
```

```
10
Xa-xa
<function newfunc.<locals>.myfunc at 0x7fbf444888b0>
389
```

Функции

Функция может принимать произвольное количество аргументов или не иметь их вовсе. Так же распространены функции с произвольным числом аргументов, функции с позиционными и именованными аргументами, обязательными и необязательными.

```
def func(a, b, c=2):          # c - необязательный аргумент
return (a + b)**c
print(func(1.1, 4.2))         # c = 2
print(func(a=2, b=-0.5))      # c = 2
print(func(a=3, c=6))         # b не определен
```

```
28.090000000000007
2.25
Traceback (most recent call last):
  File "main.py", line 5, in <module>
    print(func(a=3, c=6)) #  b не
TypeError: func() missing 1 required positional
 argument: 'b'
```

Функции

Функция может принимать переменное количество позиционных аргументов, тогда перед именем ставится* .

Функция может принимать произвольное число именованных аргументов, тогда перед именем ставится**.

```
def fun1(*ar):
s=""
for i in ar:
s+=str(i)
return s
def fun2(**kar):
return kar

print(fun1(11, "w2", 3.1, 'abc'))
print(fun1(11))
print(fun2(a="2", b=-2, c=5.11))
print(fun2())
print(fun2(a="Hellow"))
```

```
11w23.1abc
11
{'a': '2', 'b': -2, 'c': 5.11}
{}
{'a': 'Hellow'}
```

Анонимные функции. Инструкция lambda

Анонимные функции создаются с помощью инструкции lambda. Анонимные функции не имеют имени, а содержат только выражение. Однако выполняются они быстрее. lambda функции, в отличие от обычных, не требуют инструкция return:

```
fun=lambda x, a=-5,y=6: x*x - a*x + y
a=float(input("a= "))
b=float(input("b= "))
c=float(input("c= "))
print(fun(a,b,c))
for x in range(-8,6,2):
print("x=", x/2," f=",fun(x/2))
```

```
a= -2
b= 4
c= -12
0.0
x= -4.0   f= 2.0
x= -3.0   f= 0.0
x= -2.0   f= 0.0
x= -1.0   f= 2.0
x= 0.0   f= 6.0
x= 1.0   f= 12.0
x= 2.0   f= 20.0
```

Рекурсивные функции

Рекурсия — это такой способ организации вспомогательного алгоритма (подпрограммы), при котором эта подпрограмма (функция) в ходе выполнения ее операторов обращается сама к себе. Рекурсивным называется любой объект, который частично определяется через себя. В рекурсивном определении должно присутствовать ограничение, граничное условие, при выходе на которое дальнейшая инициация рекурсивных обращений прекращается.

Рекурсивные версии большинства подпрограмм могут выполняться немного медленнее, чем их итеративные эквиваленты, поскольку к необходимым действиям добавляются вызовы функций. Но в большинстве случаев это не имеет значения. Много рекурсивных вызовов в функции может привести к переполнению памяти.

Основным преимуществом применения рекурсивных функций является использование их для более простого создания версии некоторых алгоритмов по сравнению с итеративными эквивалентами.

Рекурсия бывает прямая и косвенная.

Рекурсивные функции

Пример (прямая рекурсия)

Вычисление факториала числа **n**.

```
def fak(n):
    if n==0:
        return 1
    else:
        return n*fak(n-1)
print(" n=",5," Факториал=", fak(5))
print(" n=",8," Факториал=", fak(8))
```

```
n= 5  Факториал= 120
n= 8  Факториал= 40320
```

Пример (прямая рекурсия)

Вычисления **n**-го числа Фибоначчи.

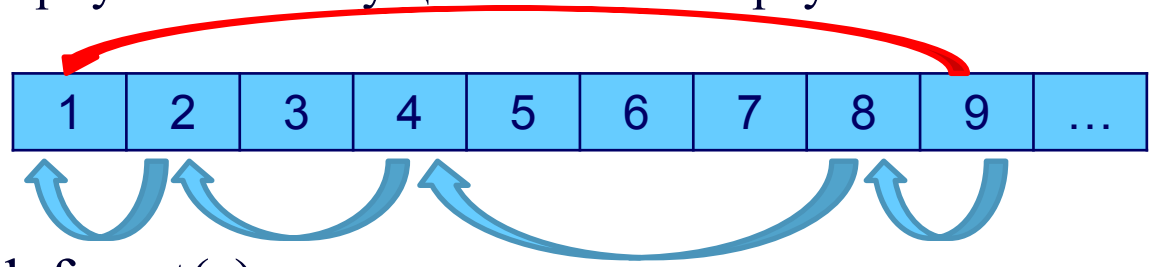
Если нулевой элемент последовательности равен 0, первый – 1, а каждый последующий равен сумме двух предыдущих, то ряд Фибоначчи (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...).

```
def fib(n):
if n==0 or n==1:
return n
else:
return fib(n-1)+fib(n-2)
print(" n=",5," число Фибоначчи=", fib(5))
print(" n=",10," число Фибоначчи=", fib(10))
```

```
n= 5  число Фибоначчи= 5
n= 10  число Фибоначчи= 55
```

Рекурсия

Рассмотрим игру. Игрок находится в произвольной клетке на пронумерованном поле. Цель вернуться в клетку №1. Если игрок находится на чётной клетке, он платит одну монету и проходит половину пути к клетке №1. Если игрок находится на нечётной клетке, он может заплатить 5 монет и сразу перейти на первую клетку или заплатить одну монету и сделать один шаг к клетке №1 - на чётную клетку. Вопрос заключается в следующем: какое наименьшее количество монет необходимо заплатить, чтобы вернуться из текущей клетки в первую.




```
def cost(s):
    if s <= 1:
        return 0
    if s % 2 == 0:
        return 1 + cost(s // 2)
    return min(1 + cost(s - 1),5)
for i in range(1,22,1):
     print(" s=",i," p=", cost(i))
```

```
s= 2   p= 1
s= 3   p= 2
s= 4   p= 2
s= 5   p= 3
s= 6   p= 3
s= 7   p= 4
s= 8   p= 3
s= 9   p= 4
s= 10   p= 4
s= 11   p= 5
s= 12   p= 4
s= 13   p= 5
s= 14   p= 5
s= 15   p= 5
s= 16   p= 4
s= 17   p= 5
s= 18   p= 5
s= 19   p= 5
s= 20   p= 5
s= 21   p= 5
```

Косвенная рекурсия

Пример

```
def f(n):
    print("f: n="," ",n,"**",end=" ")
    if n > 0: g(n - 1)
def g(n):
    print("g: n="," ",n,"&",end=" ")
     if n > 1: f(n - 2)
f(8)
```

```
f: n= 8 ** g: n= 7 &f: n= 5 ** g: n= 4 &f: n= 2 ** g: n= 1 &
```

Исключения

Для обработки особых ситуаций (таких как деление на ноль или попытка чтения из несуществующего файла) применяется механизм исключений.

Оператор try-except

```
try:
  res = int(open('a.txt').read()) / int(open('c.txt').read())
  print res
except IOError:
  print "Ошибка ввода-вывода"
except ZeroDivisionError:
  print "Деление на 0"
except KeyboardInterrupt:
  print "Прерывание с клавиатуры"
except:
  print "Ошибка"
```

В этом примере берутся числа из двух файлов и делятся одно на другое. В результате этих нехитрых действий может возникнуть несколько исключительных ситуаций, некоторые из них отмечены в частях except (здесь использованы стандартные встроенные исключения Python). Последняя часть except в этом примере улавливает все другие исключения, которые не были пойманы выше.

Исключения

Исключения (exceptions) рассматриваются как тип данных в Python. Исключения необходимы для того, чтобы сообщать об ошибках, возникающих при выполнении программы и приводящие к невозможности ее дальнейшей корректной работы.

- ArithmeticError – арифметическая ошибка
  - FloatingPointError – порождается при неудачном выполнении операции с плавающей запятой.
  - OverflowError - возникает, когда результат арифметической операции слишком велик для представления.
  - ZeroDivisionError – деление на ноль.
- MemoryError – недостаточно памяти.
- NameError – не найдено переменной с таким именем.
- RuntimeError - возникает, когда исключение не попадает ни под одну из других категорий.
- ValueError – функция получает аргумент правильного типа, но не корректного значения.

Исключения. Инструкция try-except. Инструкция raise

Простой калькулятор

```
try:
n=int(input("n="))
m=int(input("m="))
for op in
["*","+",":","div","^","/"]:
print("n",op,"m= ", end="")
if op == "+": result = n+m
elif op == "-": result = n-m
elif op == "*": result = n*m
elif op == ":": result = n/m
elif op == "div": result = n//m
elif op == "^": result = m**n
else: raise RuntimeError
    print("n",op,"m =", result)
except RuntimeError:
  print ("n",op,"m =","Калькулятор не знает этой операции ")
except ZeroDivisionError:
  print ("Деление на 0")
except KeyboardInterrupt:
  print (" Прерывание с клавиатуры") #<Ctrl>+<c>
except :
  print ("Ошибка ввода")
```

Исключения. Инструкция try-except. Инструкция raise

```
n=-11
m=5
n * m = -55
n + m = -6
n : m = -2.2
n div m = -3
n ^ m = 2.048e-08
n / m = Калькулятор не знает этой операции
```

```
n=-7
m=-4
n * m = 28
n + m = -11
n : m = 1.75
n div m = 1
n ^ m = -6.103515625e-05
n / m = Калькулятор не знает этой операции
```

```
n=5
m=7.1
Ошибка ввода
```

```
n=11
m=6    Прерывание с клавиатуры
```

```
n=12
m=0
n * m = 0
n + m = 12
Деление на 0
```

Понятие модуля

Модуль оформляется в виде отдельного файла с исходным кодом. *Подключение модуля* к программе на *Python* осуществляется с помощью оператора import

Первая форма

```
import os
import pre as re
```

Вторая форма

```
from sys import argv, environ
from string import *
```

С помощью первой формы с текущей областью видимости связывается только имя, ссылающееся на объект модуля, а при использовании второй - указанные имена (или все имена, если применена *) объектов модуля связываются с текущей областью видимости. При импорте можно изменить имя, с которым объект будет связан, с помощью as.

Понятие модуля

В первом случае пространство имен модуля остается в отдельном имени и для доступа к конкретному имени из модуля нужно применять точку. Во втором случае имена используются так, как если бы они были определены в текущем модуле:

```
os.system("dir")
digits = re.compile("\d+")
print argv[0], environ
```

Повторный импорт модуля происходит гораздо быстрее, так как модули кэшируются интерпретатором. Загруженный модуль можно загрузить еще раз (например, если модуль изменился на диске) с помощью функции reload():

```
import mymodule
. . .
reload(mymodule)
```

Встроенные функции

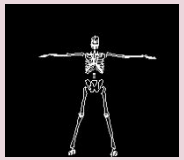
abs(x)	Модуль числа x. Результат: |x|.
divmod(x, y)	Частное и остаток от деления. Результат: (частное, остаток).
pow(x, y[, m])	Возведение x в степень y по модулю m. Результат: x**y % m.
round(n[, z])	Округление чисел до заданного знака после (или до) точки.
ord(s)	Функция возвращает код (или Unicode) заданного ей символа в односимвольной строке.
chr(n)	Возвращает строку с символом с заданным кодом.
len(s)	Возвращает число элементов последовательности или отображения.
oct(n), *hex*(n)	Функции возвращают строку с восьмеричным или шестнадцатеричным представлением целого числа n.
cmp(x, y)	Сравнение двух значений. Результат: отрицательный, ноль или положительный, в зависимости от результата сравнения.

Модули math и cmath

Функция	Описание
acos(z)	арккосинус z
asin(z)	арксинус z
atan(z)	арктангенс z
atan2(y,x)	atan(y/x)
ceil(x)	наименьшее целое, большее или равное x
cos(z)	косинус z
cosh(x)	гиперболический косинус x
e	константа e
exp(z)	экспонента (то есть, e**z)
fabs(x)	абсолютное значение x
floor(x)	наибольшее целое, меньшее или равное x
fmod(x,y)	остаток от деления x на y
log(z)	натуральный логарифм z
log10(z)	десятичный логарифм z
modf(x)	возвращает пару (y,q) - целую и дробную часть x. Обе части имеют знак исходного числа
pi	константа пи
pow(x,y)	x**y
sin(z)	синус z
sinh(z)	гиперболический синус z
sqrt(z)	корень квадратный от z
tan(z)	тангенс z
tanh(z)	гиперболический тангенс z

Модуль random

random()	Генерирует *псевдослучайное число* из полуоткрытого диапазона [0.0, 1.0).
choice(s)	Выбирает случайный элемент из последовательности s.
shuffle(s)	Размешивает элементы изменчивой последовательности s на месте.
randrange([start,] stop[, step])	Выдает случайное целое число из диапазона range(start, stop, step). Аналогично choice(range(start, stop, step)).
normalvariate(mu, sigma)	Выдает число из последовательности *нормально распределенных* псевдослучайных чисел. Здесь mu - среднее, sigma - среднеквадратическое отклонение (sigma > 0)



Обработка последовательностей

Под *последовательностью* в Python понимается любой тип данных, который поддерживает интерфейс последовательности.
Тип, основной задачей которого является хранение, манипулирование и обеспечение доступа к самостоятельным данным называется контейнерным типом или просто *контейнером*. Примеры контейнеров в Python - списки, кортежи, словари.

Функции range() и xrange()

Функция range() уже упоминалась при рассмотрении цикла for. Эта функция принимает от одного до трех аргументов. Если аргумент всего один, она генерирует список чисел от 0 (включительно) до заданного числа (исключительно). Если аргументов два, то список начинается с числа, указанного первым аргументом. Если аргументов три - третий аргумент задает шаг

```
>>> print range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> print range(1, 10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> print range(1, 10, 3)
[1, 4, 7]
```

Функция sum()

Получить сумму элементов можно с помощью функции sum():

```
>>> sum(range(10))
45
```

Эта функция работает только для числовых типов, она не может сцеплять строки.

Итераторы

Итератор представляет собой объект, который для данной последовательности выдает следующий элемент, либо генерирует исключение, если элементов больше нет.

Итераторы можно применять вместо последовательности в операторе for. Более того, внутренне оператор for запрашивает от последовательности ее *итератор*.

Функция sorted()

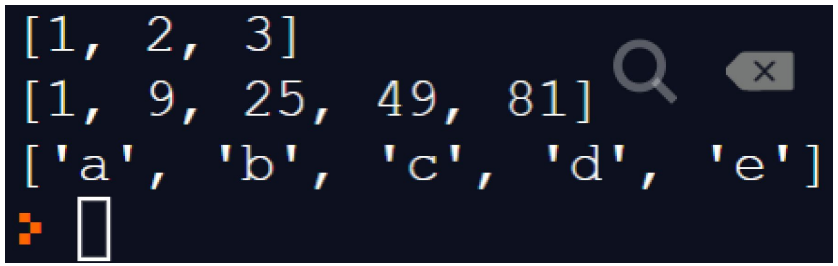
Эта функция позволяет создавать итератор, выполняющий сортировку:

```
>>> sorted('avdsdf')
['a', 'd', 'd', 'f', 's', 'v']
```

Tun list

Списки в Python – упорядоченные изменяемые последовательности объектов произвольных типов (почти как массив, но типы могут отличаться).
Их можно задать с помощью литералов, записываемых в квадратных скобках, или посредством списковых включений.

```
lst1 = [1, 2, 3]
lst2 = [x**2 for x in range(10) if x % 2 == 1]
lst3 = list("abcde")
print(lst1)
print(lst2)
print(lst3)
```



Тип tuple

Кортеж, по сути – неизменяемый список. Используется для представления константной последовательности (разнородных) объектов используется тип *кортеж*. Литерал кортежа обычно записывается в круглых скобках, но можно, если не возникают неоднозначности, писать и без них.

```
p = (1.2, 3.4, 0.9)  # точка в трехмерном пространстве
print("s= ")
for s in "one", "two", "three": print (s)
one_item = (1,)
empty = ()
p1 = 1, 3, 9 # без скобок
p2 = 3, 8, 5, # запятая в конце игнорируется
print("p= ",p)
print("s= ",s)
print(one_item)
print("p1= ",p1)
print("p2= ",p2)
p2,p1=p1,p2   # обмен значениями
print("p1= ",p1)
print("p2= ",p2)
```

```
s=
one
two
three
p=  (1.2, 3.4, 0.9)
s=  three
(1,)
p1=  (1, 3, 9)
p2=  (3, 8, 5)
p1=  (3, 8, 5)
p2=  (1, 3, 9)
```

Тип tuple

Создание кортежей

```
a = tuple() # функцией tuple()
a1 = ('s', ) # запятая обязательно!
a2 = tuple('hello, world!')
print("a= ",a)
print("a1= ",a1)
print("a2= ",a2)
```

```
a=  ()
a1=  ('s',)
a2=  ('h', 'e', 'l', 'l', 'o', ',', ' ', 'w', 'o', 'r', 'l', 'd', '!')
```

```
a1 = ('Петр ', )
a2 = tuple('hello')
print("a1= ",a1)
print("a2= ",a2)
a1,a2=a2,a1
print("a1= ",a1)
print("a2= ",a2)
```

```
a1=  ('Петр ',)
a2=  ('h', 'e', 'l', 'l', 'o')
a1=  ('h', 'e', 'l', 'l', 'o')
a2=  ('Петр ',)
```

Последовательности

Синтаксис	Семантика
len(s)	Длина последовательности s
x in s	Проверка принадлежности элемента последовательности. Возвращает True или False
x not in s	= not x in s
s + s1	Конкатенация последовательностей
s*n или n*s	Последовательность из n раз повторенной s. Если n < 0, возвращается пустая последовательность.
s[i]	Возвращает i-й элемент s или len(s)+i-й, если i < 0
s[i:j:d]	Срез из последовательности s от i до j с шагом d
min(s)	Наименьший элемент s
max(s)	Наибольший элемент s

Конструкции для изменяемых последовательностей:

s[i] = x	i -й элемент списка s заменяется на x
s[i:j:d] = t	Срез от i до j (с шагом d) заменяется на (список) t
del s[i:j:d]	Удаление элементов среза из последовательности

Некоторые методы для работы с последовательностями

Метод	Описание
.append(x)	Добавляет элемент в конец последовательности
.count(x)	Считает количество элементов, равных x
.extend(s)	Добавляет к концу последовательности последовательность s
.index(x)	Возвращает наименьшее i, такое, что s[i] == x. Возбуждает исключение ValueError, если x не найден в s
.insert(i, x)	Вставляет элемент x в i -й промежуток
.pop(i)	Возвращает i-й элемент, удаляя его из последовательности
.reverse(s)	Меняет порядок элементов s на обратный
.sort([cmpfunc])	Сортирует элементы s. Может быть указана своя *функция сравнения* cmpfunc

Взятие элемента по индексу и срезы

Для получения отдельного элемента последовательности используются квадратные скобки, в которых стоит выражение, дающее индекс. Индексы последовательностей в Python начинаются с нуля. Отрицательные индексы служат для отсчета элементов с конца последовательности (-1 - последний элемент).

Пример

```
>>> s = [0, 1, 2, 3, 4]
>>> print s[0], s[-1], s[3]
0 4 3
>>> s[2] = -2
>>> print s
[0, 1, -2, 3, 4]
>>> del s[2]
>>> print s
[0, 1, 3, 4]
```

Tun dict

Словарь (хэш, ассоциативная последовательность) - это изменчивая структура данных для хранения пар ключ-значение, где значение однозначно определяется ключом. В качестве ключа может выступать неизменчивый тип данных (число, строка, кортеж и т.п.). Порядок пар ключ-значение произволен. Ниже приведен литерал для словаря и пример работы со словарем:

```
d = {1: 'one', 2: 'two', 3: 'three', 4: 'four'}
d0 = {0: 'zero'}
print (d[1])       # берется значение по ключу
d0[0] = 0          # присваивается значение по ключу
del d0[0] # удаляется пара ключ-значение с данным ключом
print (d)
for key, val in d.items():   # цикл по всему словарю
  print (key, val)
for key in d.keys():   # цикл по ключам словаря
  print (key, d[key])
for val in d.values():   # цикл по значениям словаря
  print (val)
d.update(d0)  # пополняется словарь из другого
print (len(d)) # количество пар в словаре
```

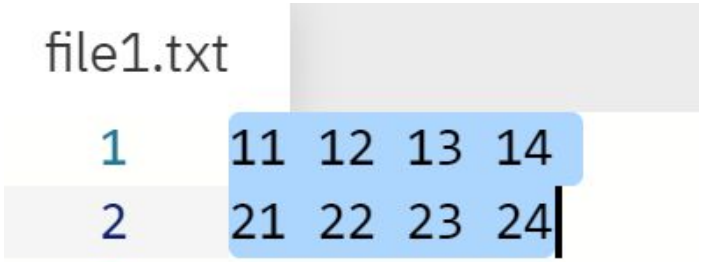
```
one
{1: 'one', 2: 'two', 3: 'three', 4: 'four'}
1 one
2 two
3 three
4 four
1 one
2 two
3 three
4 four
one
two
three
four
4
```

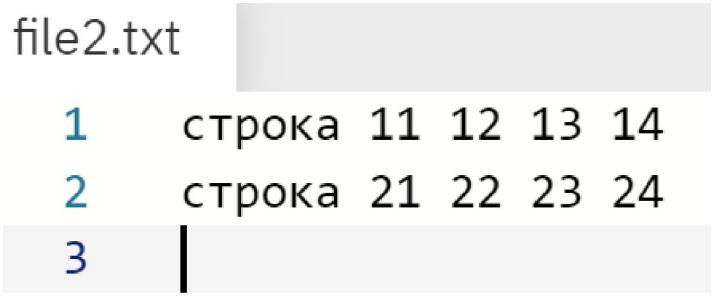
Тип file

Объекты этого типа предназначены для работы с внешними данными. В простом случае - это файл на диске. Файловые объекты поддерживают основные методы: **read(), write(), readline(), readlines(), seek(), tell(), close()**

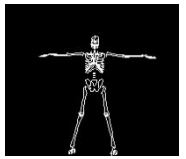
Следующий пример показывает копирование файла:

```
f1 = open("file1.txt", "r")
f2 = open("file2.txt", "w")
for z in f1.readlines():
a="строка "+z
f2.write(a)
f2.close()
f1.close()
```





Функция Аккермана



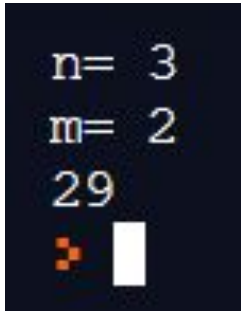
Функция Аккермана для неотрицательных чисел m и n определяется следующим образом:

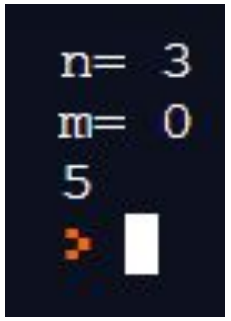
$$A(n,m)=\begin{cases} m+1, \text{ при } n=0 \\ A(n-1,1), \text{ при } n\neq 0, m=0 \\ A(n-1,A(n,m-1)), \text{ при } n>0, m\geq 0 \end{cases}$$

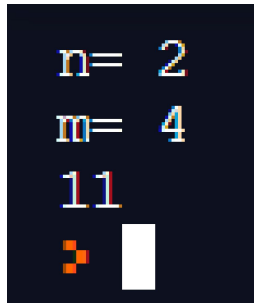
Функцию Аккермана называют “дважды рекурсивной”, так как сама функция и один из ее аргументов определены через самих себя.
Расчет значения функции Аккермана даже при небольших значениях параметров n и m требует значительных объемов памяти для размещения функции при рекурсивных вызовах.

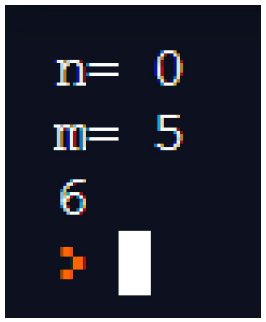
Функция Аккермана

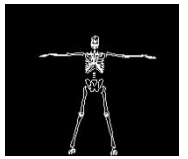
```
def Akker(n,m):
  if n==0 :
    return m+1
 elif n!=0 and m==0 :
    return Akker(n-1,1)
 elif n>0 and m>=0 :
     return Akker(n-1,Akker(n,m-1))

n=int(input("n= "))
m=int(input("m= "))
print(Akker(n,m))
```





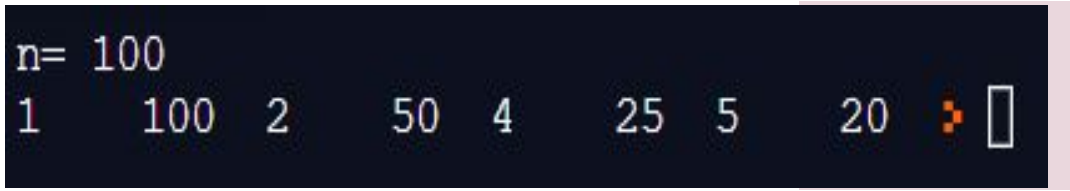


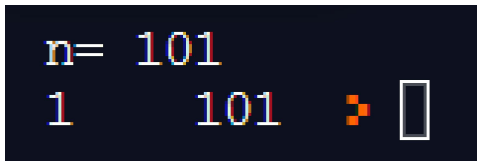




Найти все делители числа n

```
import math
def Del(n):
  for d in range (1,math.floor(n**0.5),1):
    if n%d==0:
      print(d,'  ',n//d, end='  ')
    if d**2 == n:
      print(d)

n=int(input("n= "))
Del(n)
```





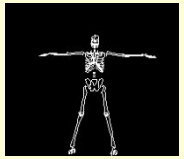
Делители не упорядочены по возрастанию

Найти все делители числа n

```
def Del1(n):
 for d in range (1,n+1):
   if d!= n % d and n % d==0:
      print(d,end=' ')

n=int(input("n= "))
Del1(n)
```

```
n= 100
1  2  4  5  10  20  25  50  100
```

Делители упорядочены по возрастанию



Спасибо за внимание