Python для новичков

Составлено по просьбе учеников школы 1392.

Шрифт сделал покрупнее, для любителей смартфонов.

Это пособие составлено для абсолютных новичков,

с самыми простыми примерами.

Удачи!

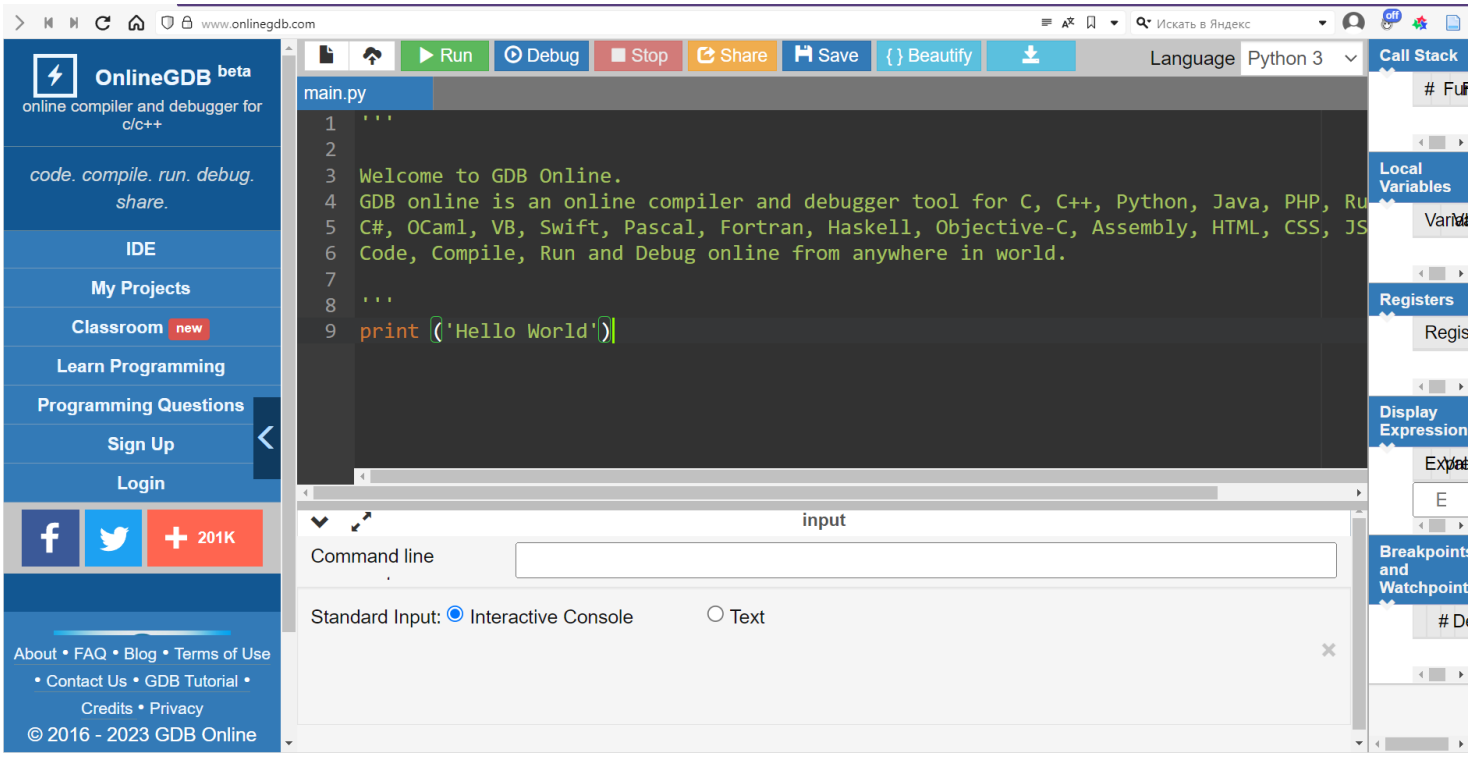
Урок 0: Подготовка к программированию.

Советую читать не с браузера, а с помощью какой-нибудь программы для PDF-файлов.

Если на вашем компьютере нет Python – не беда. В интернете есть сайты, где можно программировать онлайн. Это так называемые «песочницы». https://www.onlinegdb.com и https://replit.com — мои любимые песочницы, где можно программировать в своё удовольствие. Конечно, вы можете использовать и Python на вашем компьютере. Здесь я не буду рассказывать про установку Python.

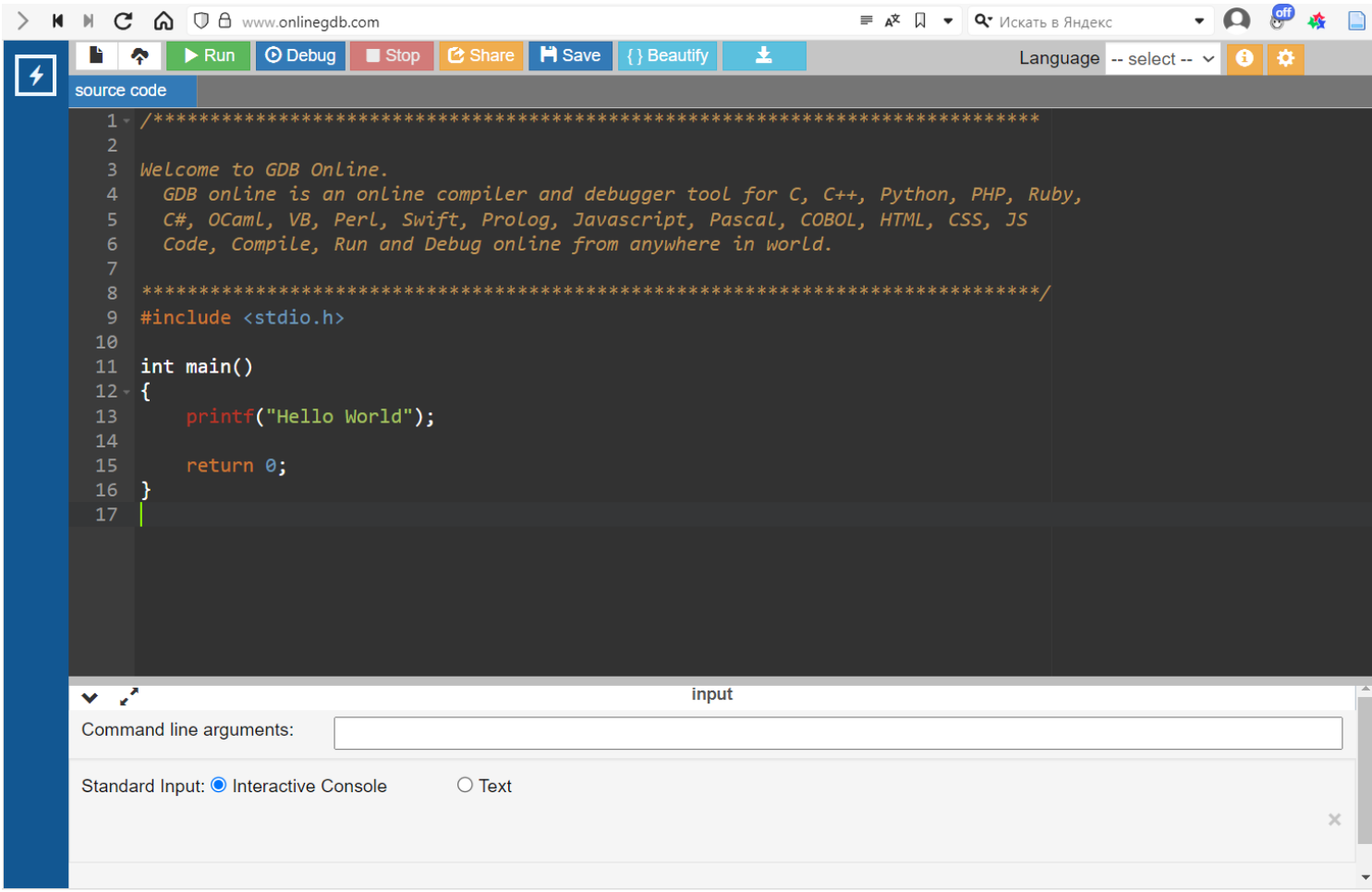
Как работать в onlinegdb:

1. Включите настольный компьютер или ноутбук (не смартфон и не планшет).
2. Зайдите на onlinegdb.com
3. Если у вас там включен автоматический переводчик с английского на русский — отключите его. Переводчиками здесь нельзя пользоваться.



4. Перед вами большое темное окно. В окне написан текст, начинающийся так: «Welcome to GDB Online. GDB online is an online compiler and debugger...». Текст в этом окне можно редактировать, как в Ворде или в Блокноте. Именно тут мы будем писать свои программы. (Обратите внимание, что строки здесь пронумерованы).
5. Настройте масштаб. Как правило, это можно сделать, зажав Ctrl и крутя колесико мыши. Если масштаб слишком большой, то некоторые кнопки могут быть не видны.
6. Настройте страничку. Разверните браузер (программу, где вы зашли на www.onlinegdb.com) на всю высоту монитора. Так вам будет лучше видно. Левую

синюю плашку можно убрать, нажав на стрелочку. Правую бело-синюю панель можно перетащить, зажав её левую границу мышкой.



7. Найдите справа наверху меню Language и выберите Python 3.
8. Сотрите всё в верхнем окне и вбейте туда свою программу, например:

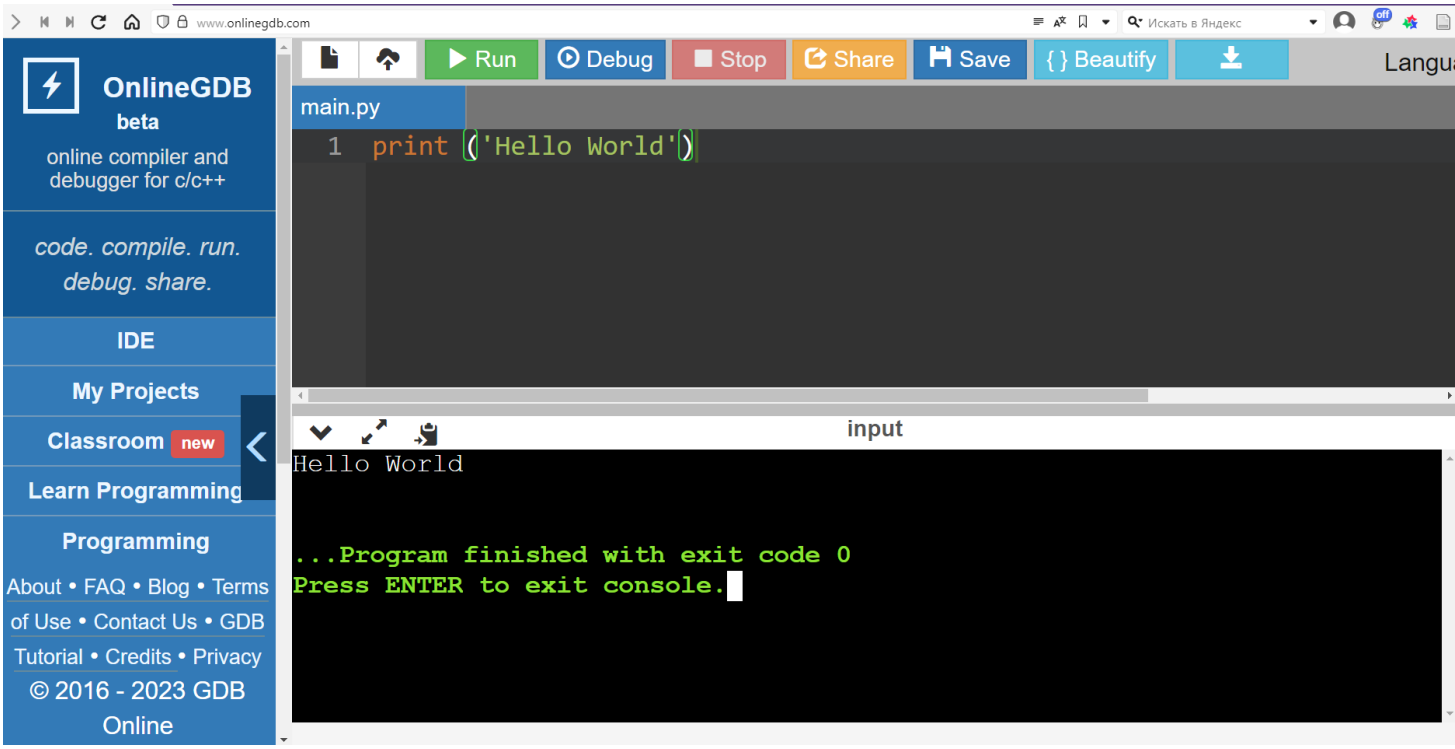
```
print(“Hello world!”)
```

Писать надо на английской раскладке. Английские кавычки на стандартной клавиатуре — Shift+Э. Когда пишете — убедитесь, что ваш курсор стоит в нужном месте.

Желательно, чтобы программа начиналась с первой строки.

Вносить код вам придется вручную, скопировать отсюда и вставить в окно не получится. Дело в том, что текстовые редакторы путают некоторые символы при переносе, например, кавычки.

9. Запустите программу. Для этого наверху есть зелёная кнопка Run.



10. Посмотрите на результат в открывшемся нижнем черном окне.

```
Hello World
```

Советую развернуть браузер на всю высоту монитора, чтобы оба окна были удобны для просмотра. Если вы наведете мышку на границу между двумя окнами, то сможете зажать и перетащить границу между ними.

Обязательно набирайте примеры на компьютере и запускайте их. Только постоянная практика позволит вам достичь успеха.

11. Программу можно сохранить прямо на сайте. Нажмите наверху оранжевую кнопку share. Вам нужна ссылка «Share Code». Когда вы нажимаете share, ваша программа автоматически сохраняется в памяти сайта. Любой человек, который перейдёт по ссылке, попадёт именно на вашу программу. Чтобы отправить программу другу или учителю, скопируйте ссылку «Share Code» и отошлите собеседнику через почту или мессенджер.

Урок 1: Вывод данных, команда print(), переменные

Когда мы запускаем любую программу — мы что-то видим на экране.

Например, это:

Мы вводим в программу данные: нажимаем на кнопки, водим пальцами по экрану.

А программа выводит нам бегущего героя, деревья, горы, количество здоровья и тд. Это называется ввод и вывод информации.

Ввод информации — это её ввод в компьютер.

Вывод информации — это её вывод из компьютера пользователю.

Поскольку мы новички, сделать сложную компьютерную игру мы не сможем. Поэтому, мы увидим в наших программах только текст и числа.

Давайте научимся выводить информацию. В языке Python это делает команда print().

Наберите эту программу в своем любимом редакторе (onlinegdb или любой другой). Данная программа занимает две строки. В языке Python на каждой строке может быть только одна команда. Запустите её.

```
print ("hello world")
print (2345)
```

Если вы всё правильно сделали, то вы должны увидеть вот это:

```
hello world
2345
```

Команда print() умеет выводить текст и числа. Кроме того, каждая новая команда выводит своё содержимое с новой строки. Обратите внимание, что текст должен быть заключен в кавычки.

Язык Python переходит на следующий абзац только, если вы напишите команду print(). Попробуйте создать пустую строку в коде и запустите программу:

```
print(“hello world”)

print(2345)
```

Вы увидите вот это:

```
hello world
2345
```

Теперь наберите эту программу (предыдущую сотрите). Здесь идёт несколько одинаковых команд подряд. Конечно, можно копировать и вставлять их в текст программы.

```
print (1625548)
print (1625548)
print (1625548)
print (1625548)
```

Запустите её и посмотрите на новый результат:

```
1625548
1625548
1625548
1625548
```

 Подобное копирование чисел довольно утомительно. К тому же, если надо будет что-то исправить — это придется делать на каждой строке.

Чтобы сэкономить усилия на исправлениях, в языках программирования были придуманы переменные. Это области памяти, куда можно поместить данные. Потом их можно использовать много раз.

Давайте создадим переменную под названием x (английская раскладка) и присвоим ей значение 1625548:

```
x=1625548456
print (x)
print (x)
print (x)
print (x)
```

Запустите её и посмотрите на результат:

```
1625548456
1625548456
1625548456
1625548456
```

Обратите внимание, мы вложили в команды print() не число, а переменную.

Давайте разберемся, что произошло. У компьютера есть память, где лежат данные. Чтобы данные не перепутались, они лежат в разных отсеках. У каждого отсека есть имя. В данном случае, Python попросил у компьютера данные из отсека «x». Компьютер нашёл у себя отсек «x» и дал Python-у его содержимое: число 1625548.

А теперь давайте поменяем значение переменной и снова запустим программу!

```
x=456
print (x)
print (x)
print (x)
print (x)
```

```

456
456
456
456
```

Значения поменялись, как по волшебству. Если вы собираетесь менять или использовать данные, то лучше это делать с помощью переменной.

В математике, значение переменной могло быть только числовым. Но, в программировании, переменная может быть и текстовой тоже.

```
x="hello world"
print (x)
print (x)
print (x)
print (x)
```

```

hello world
hello world
hello world
hello world
```

Кстати, слева и справа от знака «=» можно поставить пробелы. Это разрешено.

В языке Python мы используем латиницу. Однако, текстовые значения могут быть написаны и кириллицей:

```
x="Привет мир!"
print (x)
print ("Я уже программист!")
```

```
Привет мир!
Я уже программист!
```

Переменная может поменять своё значение прямо внутри программы. Именно поэтому она и называется «переменная». Это удобно:

```
x="Привет мир."
print (x)
print (x)
x="Я уже программист."
print (x)
print (x)
x="У меня всё получается!"
print (x)
print (x)
```

```
Привет мир.
Привет мир.
Я уже программист.
Я уже программист.
У меня всё получается!
У меня всё получается!
```

Конечно, переменных может быть несколько. И называться они могут по-разному. Одна переменная — одно слово, не больше. Переменная не может называться big apples. Зато оно может называться BigApples, bigapples, big-apples, big_apples — как вам удобнее. Главное — использовать латиницу , не начинать с цифр и учитывать заглавные и строчные буквы. Выбирайте удобные для себя имена, чтобы не забыть, для чего нужна та или иная переменная. Вот последняя программа этого урока:

```
apple=5
banan=3
orange=2

print("apple")
```

```
print(apple)
print("banan")
print(banan)
print("orange")
print(orange)

banan=0

print()

print("apple")
print(apple)
print("banan")
print(banan)
print("orange")
print(orange)
```

```
apple
5
banan
3
orange
2

apple
5
banan
0
orange
2
```

Обратите внимание, что после «обнуления» банана другие переменные не изменились. Также обратите внимание, что «пустая» команда print() создаёт пустую строку.

Не забывайте, что все примеры нужно набирать вручную. Недостаточно просто посмотреть на пример в книге.

Урок 2: Типы данных, комментарии, арифметика, конкатенация

Слово «компьютер» переводится как «вычислитель». Изначально, компьютеры были просто большими калькуляторами. Компьютер не понимает ничего, кроме чисел. Более того, компьютер понимает только единицы и нули. Его память выглядит примерно так: «...**101101010110101**...». Это двоичный код. Здесь я не буду про него рассказывать.

Однако, компьютерные программы — это не только числа. Это и текст, и звук, и фотографии. Как компьютеру удаётся с ними работать? На самом деле, он превращает всё это в двоичный код и хранит в своей памяти. Главное здесь — указать тип данных. Если компьютер ошибётся и «прочитает» фотографию как песню — возникнет ошибка.

Мы уже сталкивались с двумя типами данных — с текстом и с числами. Вообще, их больше, но изучать мы их будем постепенно. Разберёмся с ними подробнее.

Создайте две переменных — с текстом и с целым числом:

```
text="cat"
number=5
print (text)
print (number)
```

```
cat
5
```

Если бы Python не различал эти типы данных, то он не смог бы их правильно вывести.

Существует специальная команда type(). Она определяет тип данных. Увы, она не умеет выводить его на экран, поэтому нам придётся поместить её внутрь print().

Переделаем нашу программу и посмотрим, как называет типы данных сам Python:

```
text="cat"
number=5
print (type(text))
print (type(number))
```

```
<class 'str'>
<class 'int'>
```

Видно, что у этих двух переменных разные типы данных:
переменная с значением «cat» принадлежит к классу «**str**»,

переменная с значением «5» принадлежит к классу «**int**».

str – от слова «string», то есть «строка», «текст».

int – от слова «integer», то есть «целое число»

Мы уже много знаем о программировании! Мы знаем о выводе данных, о команде print(), о типах данных. Теперь мы можем сделать что-то практическое, например, посчитать:

```
print(25+31)
```

```
56
```

Python понял, что мы ввели два числа и поставили между ними «плюс». Знаки арифметики он воспринимает как команды. Поэтому, он сам их сложил. Попробуйте сами складывать и вычитать разные числа с помощью Python.

Разумеется, можно использовать не сами числа, а заранее созданные переменные:

```
x=25
y=31
print(x+y)
```

```
56
```

Зачем так делать? Python может запомнить данные только помещая их в переменные. Программы часто обмениваются данными между собой. Делают они это при помощи переменных. Поэтому, вы часто будете работать с данными внутри переменных.

Помните, что переменная — это именованная область памяти. Компьютер не запомнит информацию, если она не имеет «ярлычка». «Ярлычок» - это как раз имя переменной.

Конечно, иногда нужно сохранить результат как отдельную переменную. Давайте сделаем что-то из реальной жизни:

```
bananas_box_1=25
bananas_box_2=31
all_bananas=bananas_box_1+bananas_box_2
print(all_bananas)
```

```
56
```

Теперь переменная all_bananas содержит в себе сумму всех бананов. Кстати, попробуйте изменить значения переменных bananas_box_1 и/или bananas_box_2 и перезапустить программу:

```
bananas_box_1=235
bananas_box_2=31
all_bananas=bananas_box_1+bananas_box_2
print(all_bananas)
```

```
266
```

Эта программа считает сумму бананов из обоих ящиков. Названия переменных не дадут вам забыть, что они означают. Если количество бананов в любом ящике изменится — изменится и общий результат. Мы автоматизировали процесс подсчётов.

Так можно делать довольно большие и сложные программы. Эту программу **не нужно набирать**, просто посмотрите на неё и поймите, как она работает. Обратите внимание, здесь используются комментарии. Комментарии — это часть кода, которую Python не видит. Там можно писать заметки, напоминания, подсказки и памятки для себя и для других программистов. Многие люди оставляют похожие пометки на полях, когда читают книгу. Комментарий начинается с знака # и заканчивается концом строки.

```
#коробки с бананами
bananas_box_1=235
bananas_box_2=31

#коробки с яблоками
apples_box_1=52
apples_box_2=4
apples_box_3=52

#всего бананов
all_bananas=bananas_box_1+bananas_box_2
#всего яблок
all_apples=apples_box_1+apples_box_2+apples_box_3

print(all_bananas)
print(all_apples)

#всего яблок
all_fruits=all_bananas+all_apples

print(all_fruits)
```

```
266
108
374
```

У команды print() есть удобный инструмент для вывода текстов и чисел. Это множественный вывод, или вывод через запятую. Наберите этот пример:

```
bananas_box_1=235
bananas_box_2=31
all_bananas=bananas_box_1+bananas_box_2
print("Всего бананов:", all_bananas,"(из обоих ящиков)")
```

```
Всего бананов: 266 (из обоих ящиков)
```

Это удобно, так как занимает меньше места и лучше смотрится.
Кстати, именно так рассылаются надоедливые рекламные объявления. Посмотрите этот пример (не нужно его набирать):

```
name="Галина"
print("Здравствуйте,",name,"! Сегодня мы предлагаем вам ... ")
```

```
Здравствуйте, Галина ! Сегодня мы предлагаем вам ...
```

Мы успешно складываем числа. Но что произойдёт, если мы сложим куски текста?

```
x="Hello"
y="World"
print(x,y)
print(x+y)
```

```
Hello World
HelloWorld
```

В первом print() мы выводим обе переменных через запятую. Python вежливо ставит пробел, чтобы нам было удобнее. Во втором print() мы «склеили» два слова и получили новое - «HelloWorld». Это называется «**конкатенация**» или «склеивание». Это позволяет навсегда соединить несколько кусков текста. Например, именно так буквы соединяются в слова. Когда мы их печатаем, конкатенация происходит автоматически.

Теперь нам осталось разобраться с умножением и делением. Для этого нам понадобится ещё один тип данных: дробные числа. В отличие от уроков математики, здесь нельзя использовать запятую. Используйте точку. Наберите пример:

```
x=10
y=10.5
print(type(x))
print(type(y))
```

```
<class 'int'>
```

```
<class 'float'>
```

переменная с значением «10.5» принадлежит к классу «**float**».

float – от слова «float», то есть «плавающий». Имеется в виду выражение «числа с плавающей точкой». Так в Python называют все дробные числа из-за особенностей их записи в двоичном коде внутри компьютера. Кстати, в математике такие числа называют вещественными.

С дробными (вещественными) числами нужно быть очень осторожными. Дело в том, что Python не понимает, что 10.0 и 10 — это одно и то же. Поэтому в программе может возникнуть ошибка. Не пользуйтесь дробными числами без нужды.

Если сложить (вычесть/умножить/разделить) целое и дробное число, результат будет дробным. Наберите этот пример:

```
x=10
y=10.5
z=x+y
print(z)
print(type(z))
```

```
20.5
<class 'float'>
```

В случае деления, результат всегда будет дробным. Даже, если он обязан быть целым. В теории, 4:2=2. Однако, результат будет 2.0 Посмотрите на этот пример (не надо его набирать). Обратите внимание, для деления используется знак / .

```
x=4/2
print(x)
print(type(x))
```

```
2.0
<class 'float'>
```

Теперь вы умеете считать на языке Python. Здесь действуют те же правила, которые вы учили в начальной школе. Например, сначала считается умножение и деление, а потом — сложение и вычитание. Наберите пример:

```
x=3+4/2
print(x)
```

```
5.0
```

Урок 3: Пробелы и переносы строк в Python. Целочисленное деление, остаток.

Пробелы и переносы строк

Пробелы существуют для того, чтобы разделять слова. Когда нам хочется, мы можем поставить несколько пробелов:

например вот так

Python позволяет нам использовать пробелы, чтобы разделять служебные слова. От лишнего пробела ничего не поменяется:

```
name="Sergey"
print   (   "Hello"  ,  name  ,  "!"  )
print("Hello",name,"!")
```

```
Hello Sergey !
Hello Sergey !
```

Заметим, что пробелы перед началом команды могут испортить программу. Мы познакомимся с этим позже.

Однако, Python серьёзно относится к пробелам внутри кавычек. Для него пробел — такой же символ, как буквы, знаки препинания и цифры. Поэтому, если мы хотим **показать** пробел, его нужно заключать в кавычки, как и любой другой текст.

Если мы хотим **видеть** пробел в результате выполнения программы, то он должен быть в кавычках:

```
print("H e l l o")
print("Hello")
```

```
H e l l o
Hello
```

Если мы хотим разделить пробелами выводимые числа, то пробелы нужно вставлять вручную, в кавычках, как куски текста:

```
x=5
print(x,x,x)
print(x," ",x," ",x)
```

```
5 5 5
5   5   5
```

Теперь поговорим о переносах строк. Мы переносим строку клавишей Enter. Python разделяет свои команды именно так: каждая команда — на новой строчке. Python позволяет нам использовать Enter, чтобы разделять служебные слова. От лишнего Enter ничего не поменяется:

```
print("Hello")
print("Hello")

print("Hello")
print("Hello")
```

```
Hello
Hello
Hello
Hello
```

Каждая команда print() включает в себя перенос строки. Если мы хотим лишний раз перенести строку — нужно вставить пустую команду print():

```
print("Hello")
print("Hello")
print()
print("Hello")
```

```
Hello
Hello

Hello
```

Есть и другой способ. Поставьте прямо в тексте особый непечатаемый символ «\n».

Ставьте его там, где вы хотите оборвать строку. Python поймёт, что тут конец строки и сделает всё сам:

```
print("один\nдва\nтри")
```

```
один
два
три
```

Про типы ответов

Язык Python очень боится неверно растолковать данные. Из-за этого, иногда возникают смешные ситуации. Попробуем решить три примера, чтобы это понять.

1. Сложим 8 и 2:

```
x=8
y=2
z=x+y
print (x,type(x),y,type(y),z,type(z))
```
```
8 <class 'int'> 2 <class 'int'> 10 <class 'int'>
```
Мы сложили два целых числа (**int**) и получили целый ответ. Пока всё логично.

2. Теперь сложим дробное(**float**) и целое число:

```
x=8.5
y=2
z=x+y
print (x,type(x)," ",y,type(y)," ",z,type(z))
```
```
8.5 <class 'float'>   2 <class 'int'>   10.5 <class
'float'>
```
Мы сложили дробное и целое и получили дробный ответ. Тут тоже всё верно, ведь 10,5 — дробное число.

3. Теперь сложим два дробных числа, чтобы получить целое:

```
x=8.5
y=2.5
z=x+y
print (x,type(x)," ",y,type(y)," ",z,type(z))
```
```
8.5 <class 'float'>   2.5 <class 'float'>   11.0
<class 'float'>
```
Мы сложили два дробных числа, но получили дробный ответ. Почему так? Ведь 11 должно быть целым числом!
Как мы знаем, Python боится неверно записать данные. Дело в том, что любое целое число можно записать как дробное (11 => 11.00). А вот наоборот не получится. Поэтому, когда Python подозревает, что результат может быть дробным — он пишет не в формате int, а в формате float. Ещё раз — если Python предвидит *возможность* появления дроби, то сохраняет ответ как дробь.

На самом деле, это не очень хорошо. Дробные числа считаются гораздо медленнее и занимают больше места в памяти. Зато мы не испортим программу. Затолкать дробное число в отсек памяти для целого числа можно. Но этим мы испортим число. Например, 2.5 превратится в 2

Дело в том, что есть два разных механизма работы с числами. Механизм для дробных чисел умеет обращаться с запятой и дробной частью. Механизм для целых чисел этого не умеет. Поэтому, если к нему попадёт 2,5 , то механизм проигнорирует дробную часть.

Попробуем разделить 8 на 2:

```
x=8
y=2
z=x/y
print (z,type(z))
```

```
4.0 <class 'float'>
```

Мы получили 4, но оно оказалось дробным. Python заранее предвидит возможность появления дроби и сохраняет ответ как дробь.

Целочисленное деление

Иногда, нам не нужна дробная часть ответа. Например, мы хотим знать, сколько палет можно загрузить в грузовик «Газель». Точный ответ — 4.6. Но такой точный ответ нам не нужен. Мы можем загрузить в «Газель» только 4 палеты. Жалко, что там останется свободное место. Но пятую палету туда уже не втиснуть.

Если мы спросим шофёра, сколько палет входит в «Газель», он ответит: «четыре».

Чем длиннее грузовик, тем больше палет в него вместится. Как это выяснить? Нужно длину грузовика разделить на длину палеты.

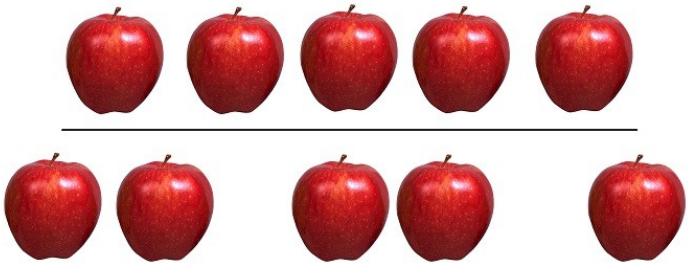
Такое деление называют «целочисленное деление» или «деление с остатком».

Когда вы делите яблоки, вам иногда приходится указывать дробную часть. Но в начальной школе, когда вы были маленькими, вы не знали дробей и делили иначе.

Давайте разделим 5 яблок на 2 части обоими способами:

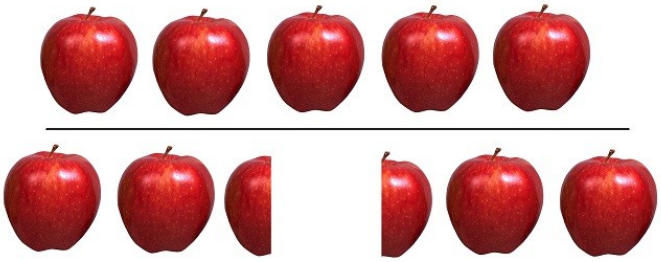
5 : 2 =

Мы ещё не знаем дробей:



5 : 2 = 2 , остаток 1

Мы уже знаем дроби:



5 : 2 = 2,5

Слева целочисленное деление. Справа обычное деление

Теперь посмотрим, как это делается в Python. Целочисленное деление в Python обозначается двойным слэшем //. Другое название - «двойная косая черта».

```
apples=5
result_1=apples/2
result_2=apples//2
print(result_1)
print(result_2)
```

```
2.5
2
```

В Python есть даже специальный знак % для нахождения остатка от деления.

Внимание, знак % в Python не обозначает процент! Это инструмент для нахождения остатка. Здесь мы выясняем, что останется, если 5 поделить на 2?

```
apples=5
ostatok=apples%2
print(ostatok)
```

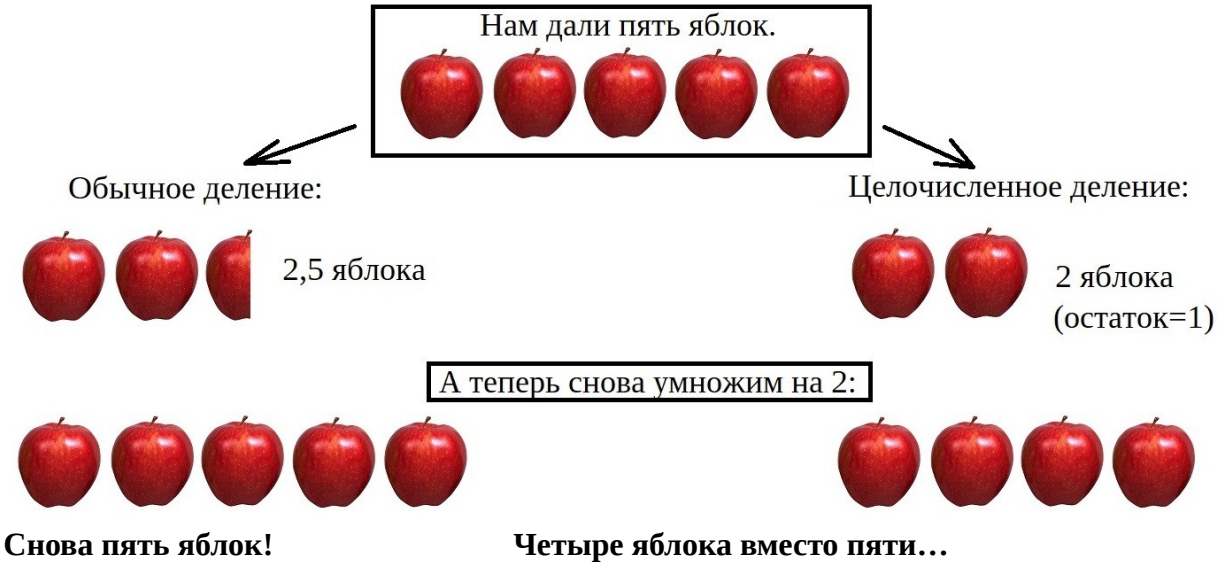
```
1
```

Действительно, при целочисленном делении 5 на 2, остаток=1.

Можно ли найти остаток, не пользуясь этим инструментом? Конечно.

Обычно, если мы делим что-то на 2, а потом умножаем на 2, результат совпадает.

Но, в случае с целочисленным делением, так бывает не всегда.



Теперь решим эту же задачу в Python. Напомним, что знак умножения в Python - “**”, а знак # - это начало комментария, который Python не «видит».

```
primer_1=5/2*2   # Тут мы делим точно пополам: 5/2=2,5
primer_2=5//2*2  # Тут мы делим нацело, без остатка: 5//2=2
print(primer_1)  # 2,5*2=5
print(primer_2)  # 2*2=4
print(5-primer_2)  #Остаток от целочисленного деления 5 на 2
```

```
5.0
4
1
```

Урок 4: Условия. Операторы сравнивания

До сих пор, Python выполнял все наши команды. Однако, в реальности, бывают ситуации, когда мы не знаем, придётся ли выполнять команду? Нужно ли автобусу остановиться на остановке «по требованию»? Должна ли стиральная машина сушить бельё после стирки? Можно ли заменить оружие герою в компьютерной игре?

Всегда есть проблема условий и дальнейших действий.

Если впереди поворот, нужно будет свернуть. **Если** надо высушить бельё, стиральная машина должна слить воду. **Если** мы хотим взять меч, то надо сперва выпустить из рук дубинку.

В программировании, условие — это, как правило, определенное значение переменной.

В Python есть служебное слово **if** (“если” на английском языке). С его помощью можно создавать условия выполнения команд. Например, что у нас за фрукт?

```
fruit="яблоко"
if fruit=="яблоко":
    print ("Хороший выбор!")
```

```
Хороший выбор!
```

Ещё пример:

```
otmetka=5
if otmetka==5:
    print ("отличник!")
```

```
отличник!
```

Внимание: команда print() сработает только при выполнении условия if.
Обратите внимание на отступ. Команда print() подчинена условию if. По правилам Python, включенные команды пишутся с отступом. По традиции, это четыре пробела или клавиша Tab.
Обратите внимание, мы использовали знак «двойное равно» (==).
Когда мы пишем fruit="яблоко", мы приказываем переменной fruit принять значение "яблоко".
Если же мы пишем fruit=="яблоко", то мы сравниваем два объекта: переменную fruit и текст "яблоко".

Знак «=» - это оператор присваивания. Он присваивает переменной значение.
Вот короткий пример:

```
fruit="яблоко"
```

Знак «==» - это оператор сравнивания. Он сравнивает два объекта и говорит конструкции if, действительно ли они равны. Ответ может быть только «да» или «нет» («истина» или «ложь»). Тут мы обсуждаем, сравниваем, *но ничего не меняем*.

Теперь попробуем примеры посложнее:

```
apples=3
if apples==3:
    print ("у нас есть три яблока!")
```

```
у нас есть три яблока!
```

Python увидел, что переменная apples равна **3**. После этого, Python запустил команду print().

```
apples=3
if apples==4:
    print ("у нас есть четыре яблока!")
```

```

```

Python увидел, что переменная apples *не равна* **4**. После этого, Python *не запустил* команду print(). Здесь *не выполняется условие*.

На самом деле, Python не понимает текст. Он не сможет вас поправить, если вы что-то неправильно напишите:

```
apples=2
```

```
if apples==2:
    print("у нас десять яблок!")
```
```
у нас десять яблок!
```
Конечно, у нас всего два яблока. Но Python не понимает человеческий язык. Он выводит неверную фразу и даже не поправит нас. Будьте внимательны. Одна из задач программиста — переводить с человеческого языка на компьютерный и наоборот.

Следующий пример:

```
apples=3
if apples==3:
    print ("у нас есть три яблока!")
    print ("три яблока – это идеал!")
    print ("кроме трёх яблок, в жизни ничего не
нужно!")
```
```
у нас есть три яблока!
три яблока – это идеал!
кроме трёх яблок, в жизни ничего не нужно!
```
Здесь у нас целый блок команд внутри конструкции **if**. Разумеется, если количество яблок неправильное, то этот блок не исполнится. Обратите внимание — весь блок имеет отступ. Это значит, что он подчинён конструкции **if**.

В программе может быть несколько конструкций **if**:

```
otmetka=5
if otmetka==5:
    print ("отличник!")
if otmetka==4:
    print ("хорошист")
if otmetka==3:
    print ("троечник")
```
```
отличник!
```
Попробуйте изменить значение otmetka. Обратите внимание, если значение будет не 5, не 4 и не 3, то ни одна конструкция if не сработает.

В условии может быть не только оператор «==». Также используются «<» и «>» («меньше» и «больше»)

```
otmetka=4
if otmetka>2:
    print ("вы не двоечник!")
```
```
вы не двоечник!
```
В данном случае, условие будет выполнено, если otmetka больше **2**. То есть, можно ставить любое значение, от 2 до бесконечности (постарайтесь не вводить очень большие числа, вроде 99999999999999999999999999999999999999999999999999999999,
Python этого не любит).

Кроме этого, можно использовать операторы «>=» и «<=»:

```
otmetka=3
if otmetka<=4:
    print ("вы не отличник")
```
```
вы не отличник
```
Здесь, чтобы условие исполнилось, переменная otmetka должна быть меньше или равна **4**.

if нужен, если мы не знаем заранее, что именно должны сделать. Поэтому, мы заранее пишем команды на случай, если условие сбудется. На самом деле, этого может и не случиться. Но нас в этот момент не будет рядом. Компьютер сам выберет, запускать ли блок команд внутри **if**.

Напоследок, познакомимся с оператором неравенства «**!=**», «не равно». Это условие выполняется, если объекты не равны между собой. «!=» - противоположность «==».
Посмотрим на последний пример этого урока:

```
otmetka=3

if otmetka!=2:
    print ("вы не двоечник")
```
```
вы не двоечник
```

Урок 5: Сложные условия. Логические операторы. Конструкция if-else.

логический оператор and.

Мы уже знаем, что бывают куски кода, которые не запустятся сами по себе. Для этого нужно исполнение условия. Если условие исполнено — код запустится.

```
lock="open"
if lock=="open":
    print("дверь открыта")
```

```
дверь открыта
```

Однако, бывают ситуации, когда у двери есть несколько замков. Например, если в двери два замка, нам нужно открыть оба. Нам недостаточно открыть один замок или другой замок. Нам нужно открыть оба замка *одновременно*.
В языке Python есть логический оператор **and**. Он позволяет применить два условия одновременно.

```
lock_1="open"
lock_2="open"
if lock_1=="open" and lock_2=="open":
    print("дверь открыта")
```

```
дверь открыта
```

Попробуйте изменить хотя бы одну переменную — и вы обнаружите, что дверь не открывается:

```
lock_1="open"
lock_2="openn"
if lock_1=="open" and lock_2=="open":
    print("дверь открыта")
```

```

```

Логический оператор **and** требует, чтобы выполнялись оба условия.

Другой пример:

```
otmetka=1
if otmetka!=2 :
    print ("вы сдали экзамен")
```

```
вы сдали экзамен
```

На самом деле, здесь автор забыл, что есть отметка «1», которая ещё хуже двойки.
Попробуем это учесть:

```
otmetka=1
```

```
if otmetka!=2 and otmetka!=1:
    print ("вы сдали экзамен")
```

Получившийся код учитывает все плохие отметки.

логический оператор or.

В некоторых ситуациях нужно, чтобы сработало *хотя бы одно* условие. Для этого используется логический оператор **or**:

```
lock_1="close"
lock_2="open"
if lock_1=="close" or lock_2=="close":
    print("дверь закрыта")
```

```
дверь закрыта
```

Попробуйте разные варианты положения замков. Если один или два замка закрыты — дверь будет закрыта. Только открытие обоих замков сделает дверь открытой.

логический тип данных bool и логический оператор not.

Многое в нашем мире можно выразить численно. 30 градусов жары, 200 грамм масла, 10 яблок, 9% уксус и так далее. Однако, иногда нам нужен простой ответ: «да» или «нет». Та дверь заперта, или нет? Есть ли у собак крылья, или нет? Мы не можем сказать, что собака имеет крылья на 35%.

В примере с дверями, мы писали «close» или «open». Однако, это не самый лучший вариант. Текст занимает довольно много места в памяти компьютера. В нашей программе надо было всегда помнить, что закрытая дверь — это close (а не «closed», 0, «zakrito», -1, «locked», «заперто» и тд).

В языке Python существует особый тип данных bool. Это не числа, не текст, а всего два логических значения: True и False (в переводе с англ., «Истина» и «Ложь»). Чем они лучше «close» или «open»? Они признаны всеми программистами мира и почти не занимают места в памяти компьютера.
Но главное — Python умеет превращать True и False друг в друга с помощью логического оператора **not**.

Посмотрим, как выглядит пример с замками и дверью:

```
lock_1_is_open=True
lock_2_is_open=True
if lock_1_is_open==True and lock_2_is_open==True:
    print("дверь открыта")
```

```
дверь открыта
```

Обратите внимание, переменные пришлось переписать. Слова «close» и «open» были понятнее для человека, чем True и False. Зато теперь нам доступна упрощенная форма записи:

```
lock_1_is_open=True
lock_2_is_open=True
if lock_1_is_open and lock_2_is_open:
    print("дверь открыта")
```
```
дверь открыта
```
Это подарок от создателей Python. Теперь, когда Python понимает, что речь идёт о логических значениях (истина или ложь), он может использовать сокращенный синтаксис.

Теперь познакомимся с логическим оператором **not**. Он меняет объект на его противоположность. Это великолепно работает с логическим типом данных, так как у них всего два значения: True и False. Давайте изменим какое-нибудь значение с помощью not:

```
lock_is_open=True
print( lock_is_open)
print( lock_is_open)
print( lock_is_open)
lock_is_open = not lock_is_open
print( lock_is_open)
print( lock_is_open)
print( lock_is_open)
```
```
True
True
True
False
False
False
```
Обратите внимание, здесь раз за разом выводится переменная lock_is_open.
Однако, команда lock_is_open = not lock_is_open меняет значение переменной.
Поэтому, мы видим, что True стала False.
Попробуйте использовать 2-3 команды lock_is_open = not lock_is_open подряд. Посмотрите на результат. Переменная будет изменяться «туда-сюда» каждый раз, когда вы ей приказываете.

```
 lock_is_open=True
print( lock_is_open)
lock_is_open = not lock_is_open
print( lock_is_open)
lock_is_open = not lock_is_open
print( lock_is_open)
```
```
True
False
True
```

Такие переменные нужны, чтобы отслеживать состояния «да или нет». Например, если ученик Иванов прогуливает школу, то Ivanov=False. А если он пришёл и приложил пропуск, то Ivanov=True. Примерно так работает система учёта посещаемости в школах.

Логический оператор not превращает объект в его противоположность. В случае с «closed» это было невозможно. Во что его превращать? В «open», «открыто», «не_заперто»? Python не понимает человеческий язык. Зато он отлично понимает True и False.

Можно ли сохранить результат логического высказывания в переменную? Конечно:

```
lock_1_is_open=False
lock_2_is_open=True
door_open=lock_1_is_open and lock_2_is_open
print("door_open:",door_open)
```

```
door_open: False
```

Напоминаем, door_open – это переменная. А «door_open» - это пояснительный текст. Попробуйте поменять значения замков и посмотрите на результат.

В завершение этой темы, попробуем кое-что посложнее:

```
lock_1_is_open=False
lock_2_is_open=True
door_open=lock_1_is_open and lock_2_is_open

if door_open:
    print("дверь открыта")

if not door_open:
    print("дверь закрыта")
```

```
дверь закрыта
```

Внимательно разберите этот пример. Сравните его с первыми примерами в этом уроке.

конструкция if-else.

Мы уже знакомы с конструкцией if. Если исполняется условие — исполняется дополнительный блок кода:

```
lock_is_open=True
print("Перед нами дверь квартиры")
if lock_is_open:
    print("дверь открыта")
print("Это новенькая стальная дверь")
```

```
Перед нами дверь квартиры
```

```
дверь открыта
Это новенькая стальная дверь
```

1-я и 3-я фразы будут напечатаны в любом случае, а 2-я — только при соблюдении условия (см. урок 4, отступы кода).

Часто бывает, что мы должны отреагировать на оба варианта развития событий: и на открытую дверь, и на закрытую дверь:

```
lock_is_open=False
if lock_is_open:
    print("дверь открыта")
if not lock_is_open:
    print("дверь закрыта")
```

```
дверь закрыта
```

Python обязательно проверит *обе* конструкции if, но выполнит только одно, ибо их условия противоположны. Вопрос — нужно ли в этом случае проверять *оба* **if**?

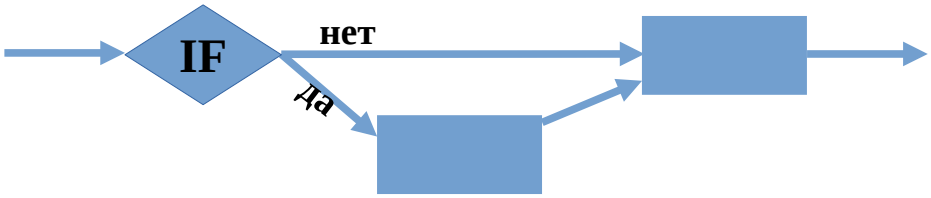
Существует особая форма **if-else**, которая изначально учитывает оба варианта развития событий:

```
lock_is_open=False
if lock_is_open:
    print("дверь открыта")
else:
    print("дверь закрыта")
```

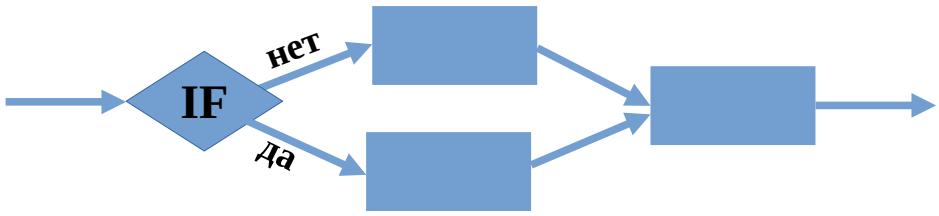
```
дверь закрыта
```

Это не две конструкции (if и else). Это одна конструкция «**if-else**». Она была создана специально для противоположных условий. Она отнимает у компьютера меньше сил и памяти.
Python проверит условие. Но, если оно не выполнено, *он не будет проверять второе условие*. Если первое условие не сработало, значит, второе обязательно сработает. Обратите внимание, второе условие даже не написано, Python о нем вообще не знает. Он просто выполнит второй блок кода, если первое условие не сработало.

Если нарисовать схему, то обычный if выглядит так ():



А схема if-else выглядит так:



Конечно, нельзя использовать if-else во всех случаях:

```
lock_is_open=False
apples=0

if lock_is_open:
    print("дверь открыта")

if apples≤0:
    print("у нас нет яблок")
```

```
у нас нет яблок
```

Здесь два условия вообще не связаны друг с другом. Так поступать не надо.

Давайте посмотрим последний пример:

```
apples=20

if apples>10:
    print("у нас много яблок")
else:
    print("у нас мало яблок, или их вообще нет")
```

```
у нас много яблок
```

Помните, что Python не понимает человеческий язык. Это не Python решил, что 11 яблок — это много. Это мы так решили.

Раньше, когда мы реагировали на два варианта развития событий, мы создавали две конструкции if. Теперь мы можем создавать одну конструкцию if-else. Это экономит память, делает код понятнее и короче. Со временем, вы привыкните.

Урок 6: Сложные условия. Полная конструкция if-elif-else.

Оптимизация выбора.

Иногда проблема выбора сложнее, чем «да» или «нет». Например, выбор автомобиля. Если у вас много денег, надо брать «Мерседес». Если денег не очень много - «Ладу». А если денег совсем мало, лучше купить велосипед. Это выбор из трёх вариантов.

Вспомните, что Python умеет выбирать только из двух вариантов. Как быть? Нужно сделать две конструкции if, одна внутри другой:

```
millions=0.6

if millions>5:
    print('покупаем Мерседес')

if millions<=5:
    if millions>=1:
        print('покупаем Ладу')
    if millions<1:
        print('покупаем велосипед')

print("Выбор сделан!")
```

```
покупаем велосипед
Выбор сделан!
```

Такие конструкции называются «вложенными условиями» (if внутри if). Внимательно изучите программу, это важно.

Теперь разберем вариант, когда у вас много денег:

```
millions=7

if millions>5:
    print('покупаем Мерседес')

if millions<=5:
    if millions>=1:
        print('покупаем Ладу')
    if millions<1:
        print('покупаем велосипед')

print("Выбор сделан!")
```

```
покупаем Мерседес
```

```
Выбор сделан!
```

Обратите внимание, насколько неэффективна такая схема. Даже, если у нас много денег, мы должны проходить через другие проверки.

Давайте постепенно перепишем этот код с применением if-else:

```
millions=0.6

if millions>5:
    print('покупаем Мерседес')

if millions≤5:
    if millions≥1:
        print('покупаем Ладу')
    else:
        print('покупаем велосипед')

print("Выбор сделан!")
```

```
покупаем велосипед
```

Теперь компьютеру стало легче считать. Для компьютера гораздо проще подставить значение из else, чем проверять ещё один if.
Возможно, эта разница вам покажется незначительной для такого простого примера. Однако, в больших программах, это даёт большую выгоду.
Ярки й пример — компьютерные игры. Пролетит ли дракон сквозь пещеру, заденет стены или застрянет намертво? Это сложные расчеты. Не хотелось бы повторять их несколько раз. Плохо сделанная игра будет «тормозить» даже на самом дорогом компьютере или смартфоне.

Следующий этап изменения кода:

```
millions=0.6

if millions>5:
    print('покупаем Мерседес')
else:
    if millions≥1:
        print('покупаем Ладу')
    else:
        print('покупаем велосипед')
print("Выбор сделан!")
```

```
покупаем велосипед
```

Теперь здесь нет ни одного лишнего if. Сперва мы проверяем самое главное — хватит ли нам денег на «Мерседес». И только, если денег маловато, мы смотрим в сторону «Лады». Велосипед остаётся на крайний случай.

Именно так строятся сложные схемы выбора.

Конструкция if-elif-else

Конечно, такие схемы тяжелы для понимания. Поэтому, в языке Python есть конструкция if-elif-else.
Слово «elif» - это сокращение от «else if», в переводе с английского «если не сработало, то, если...».

Это сокращенный синтаксис, удобный для понимания:

```
millions=0.6

if millions>5:
    print('покупаем Мерседес')
elif millions>=1:
    print('покупаем Ладу')
else:
    print('покупаем велосипед')

print("Выбор сделан!")
```

```
покупаем велосипед
Выбор сделан!
```

Итак, здесь есть только одно условие — есть ли у нас хотя бы 5 миллионов.
Если есть — покупаем Мерседес, и другие варианты компьютер рассматривать не будет.
Если денег не хватило, то нам выставят другое условие: **millions>=1**.
Если есть — покупаем Ладу. Если нет, то мы идём в else. Else – это на самый крайний случай. Здесь мы покупаем велосипед.

Таким образом можно строить сложные схемы выбора. Напоследок, приведем пример сложной схемы:

```
millions=0.6

if millions>5: print('покупаем Мерседес')
elif millions≥3: print('покупаем Рено')
elif millions≥2: print('покупаем китайский
автомобиль')
```

```
elif millions≥1: print('покупаем Ладу')
elif millions≥0.5: print('покупаем мотоцикл')
else: print('покупаем велосипед')

print("Выбор сделан!")
```

```
покупаем мотоцикл
Выбор сделан!
```

Кстати, если внутри условия всего одна команда, то строку можно не переносить. Это делает наш код гораздо компактнее и красивее.

Урок 7. Циклы. Цикл while.

Часто, нам надо много раз делать одно и то же. Мы моем тарелку за тарелкой, преодолеваем ступеньку за ступенькой. Такие действия программисты называют циклическими. Вот как это выглядит в Python:

```
print(“шаг”)
print(“шаг”)
print(“шаг”)
print(“шаг”)
```

```
шаг
шаг
шаг
шаг
```

Чем дольше нужно шагать, тем больше надо писать команд. Это неудобно.
Для автоматизации этого процесса, придумали особую конструкцию — **цикл**.
Их бывает два вида, **while** и **for**. В этом уроке мы изучим цикл while.

Цикл while

While переводится с англ. «до тех пор, пока». Мы поместим команду print(“шаг”) внутрь цикла. Там она сможет работать безостановочно. *ОСТОРОЖНО! Заранее узнайте в интернете или у учителя, как остановить программу в ВАШЕМ редакторе.* Когда вы убедитесь, что программа не собирается останавливаться, остановите её сами. В onlinegdb, например, это красная кнопка STOP.

```
x=0
while x<5:
    print(“шаг”)
```

```
...
шаг
шаг
шаг
шаг
...
```

На самом деле, слово «шаг» будет выводиться раз за разом, *бесконечно*. Это сложно показать на бумаге, легче самому набрать код и попробовать запустить.
Здесь я принудительно остановил программу. Иначе, цикл работал бы бесконечно.

Почему так произошло? Давайте «подсмотрим», что происходит с x:

```
x=0
while x<5:
    print(“шаг ”,x)
```

```
...
```

```
шаг 0
шаг 0
шаг 0
шаг 0
 ...
```

Оказывается, цикл каждый раз смотрит на переменную x, видит, что она равна нулю и смело перезапускается. Это происходит, потому что 0<5.

Попробуйте сделать x=6 и цикл вообще не запустится.
Он запускается только, если x меньше 5. Это похоже на конструкцию if, но она работает один раз, а while может работать без остановки.

Бесконечная программа - это неплохо. Такое называют «зацикливание». Но нам не всегда нужна бесконечность.

Как сделать так, чтобы цикл сработал несколько раз, а потом остановился?

Допустим, нам надо сделать пять шагов. Обычно, люди идут, считая вслух, чтобы не ошибиться. «Один, два, три, четыре, пять». На слове «пять» остановимся.
Давайте вместо x=5 напишем x=x+1. Это делает x на 1 больше.

```
x=0
while x<5:
    print(“шаг ”,x)
    x=x+1
```

```
шаг  0
шаг  1
шаг  2
шаг  3
шаг  4
```

(Почему x=0? По давней традиции, отсчёт в программировании идёт с нуля, а не с единицы).
С каждым срабатыванием цикла, x становится больше на 1.

При x=0 цикл сработает, напечатает 0 и прибавит к x единицу.
При x=1 цикл сработает, напечатает 1 и прибавит к x единицу.
При x=2 цикл сработает, напечатает 2 и прибавит к x единицу.
При x=3 цикл сработает, напечатает 3 и прибавит к x единицу.
При x=4 цикл сработает, напечатает 4 и прибавит к x единицу.
При x=5 цикл НЕ сработает, так как у него условие x<5.

Обратите внимание, мы посчитали шаги не с 1 до 5, а с 0 до 4. Но получили всё равно 5 шагов.

Мы получили замечательную универсальную конструкцию. Она работает столько раз, сколько требуется.

Главное достоинство цикла в том, что он раз за разом выполняет ЛЮБОЙ блок кода внутри себя. Если туда поместить print(“шаг”), он тоже выполнится несколько раз.

```
x=0
while x<5:
    print(“шаг ”,x)
    print(“Я цикл while, я могу повторить что угодно!”)
    x=x+1
```

```
шаг  0
Я цикл while, я могу повторить чоо угодно!
шаг  1
Я цикл while, я могу повторить что угодно!
шаг  2
Я цикл while, я могу повторить что угодно!
шаг  3
Я цикл while, я могу повторить что угодно!
шаг  4
Я цикл while, я могу повторить что угодно!
```

Обязательно попробуйте изменить количество шагов. Для этого есть много способов.

- Можно поменять изначальное значение x, например, x=3.

- Можно поменять шаг прибавки, например x=x+2.

Можно поменять условие, например, x<10.

Можно изменять все эти значения одновременно.

Цикл while - очень гибкий инструмент.

С его помощью можно моделировать разные процессы, управлять современными станками на заводах, регулировать грузовые перевозки и тд.

Рассмотрим сложный пример. С помощью цикла можно перевозить яблоки. Кстати, если ваш редактор поддерживает подсветку кода, ваша жизнь станет гораздо легче:

```
#курьер переносит ящик (на 200 яблок) за 1 рейс
#Он всегда полностью заполняет ящик (если хватает яблок)
apples=1500 #количество яблок
poezdka=0   #количество поездок

while apples > 0:
```

```
    if apples>=200: #если яблок 200 и более
        apples=apples-200
        print("набрали полный ящик яблок и
унесли")
    else:           #если яблок 199 и менее
        apples=0
        print("забрали остатки яблок и унесли")
    poezdka=poezdka+1

print("яблок осталось: ",apples)
print("поездок сделано: ", poezdka)
```

```
набрали ящик яблок и унесли
набрали ящик яблок и унесли
набрали ящик яблок и унесли
набрали ящик яблок и унесли
набрали ящик яблок и унесли
набрали ящик яблок и унесли
набрали ящик яблок и унесли
забрали остатки яблок и унесли
яблок осталось:  0
поездок сделано:  8
```

Урок 8. Цикл for. Списки.

Цикл for

Мы уже знаем, что такое цикл while. Это конструкция, которая повторяет код, пока выполняется некое условие. Однако, создавать цикл while каждый раз - утомительно. Надо заранее создавать переменную - счётчик, увеличивать её и тд.
Чтобы облегчить жизнь программиста, был создан другой цикл - цикл for. Давайте начнём разбираться. Сравним два одинаковых цикла разного типа, while и for.

```
i=0
while i<4:
  print(i)
  i=i+1
  print(i)
```

```
0
1
2
3
```

Теперь попробуем сделать то же самое, но при помощи цикла for.

```
for i in range(4):
  print(i)
```

```
0
1
2
3
```

Получилось то же самое! Конструкция for i in range(4) выглядит странно с точки зрения русского языка. Её можно перевести как «во имя i, которая подрастает от 0 до 4». Перевод звучит плохо. Поэтому, не задумывайтесь о значении этих слов. Просто помните, что тут цикл сработает 4 раза (i растёт с 0 до 4).

Ключевое слово range() показывает, до какого числа будет идти отсчёт.
Цикл for автоматизирован. Здесь переменная i создаётся сама собой. Более того, она самостоятельно увеличивается.
Конечно, это противоречит правилам создания и изменения переменных. Но создатели языка Python решили облегчить нам жизнь. Мы сэкономили две строки! Такие нарочитые нарушения со стороны создателей называются «синтаксический сахар».
Попробуйте применить цикл for несколько раз, с разным количеством повторений. Обратите внимание, что range(4) не включает в себя саму четверку, отсчёт идёт до трёх включительно.

На самом деле, цикл for - очень мощный инструмент. В современном программировании обычно используют именно его. Напишем ещё раз тот же цикл, но применим так называемую

полную форму цикла for. Это более подробный, «профессиональный» вариант цикла. Итак, давайте посмотрим на код:

```
for i in range(0,4,1):
  print(i)
```

```
0
1
2
3
```

Здесь мы видим не один, а целых 3 параметра внутри range: 0, 4, 1. Создатели языка Python отвели каждому из этих параметров особую роль. Не перепутайте их!

- 0 - это начальная точка отсчёта. В программировании мы считаем с нуля, помните?
- 4 - это конечная точка. В данном случае, мы считаем до четырёх.
- 1 - это размер прибавки. В данном случае, переменная i, с каждым разом, подрастает на 1. Вообще, это называется «шаг прибавки».

Итак, range(0,4,1) значит отсчёт от 0 до 4 с шагом 1.

Зачем нужна полная форма for? Чтобы использовать нестандартный отсчет. Например, range(3,0,-1) означает обратный отсчёт, от 3 до 0:

```
for i in range(3,0,-1):
  print(i)
```

```
3
2
1
```

Здесь мы считаем «сверху вниз», от 3 до 0. Поскольку мы не прибавляем, а вычитаем, то размер прибавки здесь равен «-1».
Другой пример: range(20,30,2) - это счёт от 20 до 30, причём, «перепрыгивая» через 1:

```
for i in range(20,30,2):
  print(i)
```

```
20
22
24
26
28
```

Попробуйте разные варианты цикла в его полной форме.

Введение в списки

Мы уже умеем сохранять различные данные внутри переменных. Например, money=5000, fruit=”яблоко”, auto=”BMW”, hot_water=True.

Однако, бывают ситуации, когда мы имеем дело с группами переменных, которые нам хочется держать вместе. Например, список продуктов: bread=1, tomato=4, milk=1, chicken_wings=10. Увы, компьютер не понимает, что эти переменные как-то связаны. Как быть?

Существует способ сохранить несколько значений в одну переменную. В языке Python есть тип данных «список» (англ. “list”). Вот как он выглядит:

```
fruits=["apple","orange","banana","mango"]
numbers=[62,84,95,13,79]
labirinth_path=["left","right","left","right","le
ft","left"]
test_answers=[True,True,False,True]

print(fruits)
print(numbers)
print(labirinth_path)
print(test_answers)
```

```
['apple', 'orange', 'banana', 'mango']
[62, 84, 95, 13, 79]
['left', 'right', 'left', 'right', 'left', 'left']
[True, True, False, True]
```

Список — отличная вещь для набора значений. Здесь мы указали:

- набор фруктов (применяется в торговле, указывает товароведу нужные товары)
- числа (применяется в гостиничном бизнесе, указывает свободные номера в отеле)
- маршрут в лабиринте (применяется в навигаторе, как голосовая помощь водителю)
- ответы на тест (применяется в образовании, определяет верные ответы на экзамене)

Кстати, в список можно вносить данные разных типов:

```
x=["apple",5,True,0.94]
```

Например, во многих тестах есть вопросы разных типов. Компьютер будет проверять правильность ответов именно таким списком.

Конечно, нам не всегда нужен *весь* список. Часто мы хотим поработать с каким-то одним значением. Как это сделать?
Когда Python создаёт список, он присваивает каждому его элементу порядковый номер, индекс. В программировании отсчёт идёт с нуля, поэтому первый элемент имеет индекс «0», второй элемент — индекс «1» и так далее.

Fruits=["apple","orange","banana","mango"]

0 1 2 3

Если мы хотим увидеть именно банан, нам нужен не просто fruits, а fruits[2]:

```
fruits=["apple","orange","banana","mango"]
print(fruits[2])
```

```
banana
```

Обратите внимание, что список создали лично мы, а пронумеровал его уже компьютер. Попробуйте вызывать разные элементы списка, в том числе, несуществующие (тут явно нет fruits[5], но вы попробуйте).

Учёт запчастей в автосервисе, список учеников в школе, количество товаров в магазине, курс доллара за последний месяц — во всех этих задачах используются списки. Это наш лучший помощник в реальном программировании.

Как изменить элемент в списке? Так же, как мы меняем обычную переменную:

```
fruits=["apple","orange","banana","mango"]
fruits[2]="cherry"
print(fruits)
```

```
['apple', 'orange', 'cherry', 'mango']
```

Бананы превратились в вишню.

Списки можно менять. Можно удалять одни элементы и добавлять другие. Для этого существуют специальные команды. Вот как можно удалить ненужный фрукт:

```
fruits=["apple","orange","banana","mango"]
fruits.remove("orange")
print(fruits)
```

```
['apple', 'banana', 'mango']
```

Мы использовали специальную команду *.remove()*, чтобы удалить “orange”.

Можно добавить ещё один фрукт:

```
fruits=["apple","banana","mango"]
fruits.append("orange")
print(fruits)
```

```
['apple', 'banana', 'mango', 'orange']
```

Мы использовали специальную команду *.append()*, чтобы добавить “orange”.
Обратите внимание, элемент добавился в конец списка.

Итак, мы научились создавать списки, изменять их, вызывать по одному элементу за раз. Теперь попробуем перебирать списки по одному элементу. Часто нам нужно сделать одно и то же с каждым элементом списка. Например, давайте повысим зарплату *всех* сотрудников на 1000 рублей:

```
salary=[20000,35000,40000,30000]
salary[0]=salary[0]+1000
salary[1]=salary[1]+1000
salary[2]=salary[2]+1000
salary[3]=salary[3]+1000
print(salary)
```
```
[21000,36000,41000,31000]
```
Получилось неплохо. Но вы заметили — мы четыре раза сделали одно и тоже. Различался только индекс, все действия были одинаковыми!

Мы знаем, что для повторяющихся действий у нас есть конструкция «цикл». Попробуем её применить. Внимательно следите за кодом. Для начала, попробуем вывести сам список несколько раз:

```
salary=[20000,35000,40000,30000]
for i in range(3):
    print(salary)
```
```
[20000, 35000, 40000, 30000]
[20000, 35000, 40000, 30000]
[20000, 35000, 40000, 30000]
```
OK, сработало. Внимание: мы применяем цикл for вместо while, чтобы код был короче и проще.

Теперь попробуем выводить не весь список, а только один его элемент:

```
salary=[20000,35000,40000,30000]

for i in range(3):
    print(salary[0])
```
```
20000
20000
20000
```
OK, сработало. Но можем ли мы выводить не один и тот же элемент, а разный? Да, можем. Для этого, нам нужно вместо salary[0] написать salary[i]. Переменная i каждый раз меняется, поэтому salary[i] тоже будет меняться.

Раньше мы три раза выводили salary[0]. Теперь же мы будем выводить salary[i].

Вот как это работает:

Наш цикл срабатывает три раза.
В первый раз, i=0. Значит, salary[i] – это salary[0].
В второй раз, i=1. Значит, salary[i] – это salary[1].
В третий раз, i=2. Значит, salary[i] – это salary[2].
Для ясности, будем выводить не только элемент массива, но и переменную i:

```
salary=[20000,35000,40000,30000]
for i in range(3):
    print(i,salary[i])
```
```
0 20000
1 35000
2 40000
```
Отлично! Но почему не вывелось последнее значение? Дело в том, что наш список содержит 4 элемента: salary[0],salary[1],salary[2],salary[3]. А вот наш цикл «растёт» с 0 до 2. Чтобы вывести **весь** список, нужно сделать цикл чуть длиннее:

```
salary=[20000,35000,40000,30000]
for i in range(4):
    print(i,salary[i])
```
```
0 20000
1 35000
2 40000
3 30000
```
Теперь мы вывели весь список. В классическом программировании это называют «перебор списка циклом». Кстати, попробуйте сделать цикл ещё длиннее, чтобы он выводил salary[4], salary[5] и так далее. Программа сломается, потому что таких элементов в нашем списке не существует.

Теперь, когда мы умеем перебирать список, давайте увеличим зарплату каждому сотруднику на 1000 рублей:

```
salary=[20000,35000,40000,30000]

for i in range(4):
    salary[i]=salary[i]+1000

print(salary)
```
```
[21000, 36000, 41000, 31000]
```
Получилось! Сегодня мы сделали большой шаг в изучении программирования. Мы умеем изменять много значений одним действием по единому шаблону. До середины 20 века, такие действия производились вручную. В любой фирме был особый отдел — «расчётное бюро», где сотрудники вручную считали подобные вещи. Программирование дало возможность автоматизировать подобные задачи.

Урок 9. Подпрограммы. Процедуры

Говорят, что каждый предмет имеет своё предназначение, свою функцию. Фломастер нужен для рисования, стул нужен, чтобы на нём сидеть, лампа нужна для освещения.

Программный код тоже создаётся с какой-то целью. Иногда мы не знаем точно, когда он нам понадобится. Тогда мы пишем его «про запас», «в долгий ящик».

Вот пример: автомобиль поворачивает налево/направо

```
#поворот направо
print ('Машина замедляется')
print ('включается поворотник')
print ('Машина поворачивает направо')
print ('Машина ускоряется','\n')

#поворот налево
print ('Машина замедляется')
print ('включается поворотник')
print ('Машина поворачивает налево')
print ('Машина ускоряется','\n')

#поворот направо
print ('Машина замедляется')
print ('включается поворотник')
print ('Машина поворачивает направо')
print ('Машина ускоряется','\n')

#поворот налево
print ('Машина замедляется')
print ('включается поворотник')
print ('Машина поворачивает налево')
print ('Машина ускоряется','\n')
```

```
Машина замедляется
включается поворотник
Машина поворачивает направо
Машина ускоряется

Машина замедляется
включается поворотник
Машина поворачивает налево
```

```
Машина ускоряется

Машина замедляется
включается поворотник
Машина поворачивает направо
Машина ускоряется

Машина замедляется
включается поворотник
Машина поворачивает налево
Машина ускоряется
```

Каждый поворот — довольно сложная штука. Нужно немного притормозить, включить сигнал поворота, повернуть, а потом чуть ускориться.

Как упростить код? Надо создать подпрограмму. Это программа внутри программы. Мы создадим кусок кода «про запас». Используйте ключевое слово def, придумайте имя для подпрограммы и поставьте после него скобки и двоеточие. Все команды подпрограммы идут с отступом.

```
def right():#поворот направо (англ. «right»)
    print ('Машина замедляется')
    print ('включается поворотник')
    print ('Машина поворачивает направо')
    print ('Машина ускоряется','\n')
```

Мы создали подпрограмму right(). Но ничего не произошло! Мы создали кусок кода «про запас», но нам надо его запустить, активировать. Для этого, нужно написать имя подпрограммы, не забыв скобки.

```
def right():#поворот направо
    print ('Машина замедляется')
    print ('включается поворотник')
    print ('Машина поворачивает направо')
    print ('Машина ускоряется')

right()
```

```
Машина замедляется
включается поворотник
Машина поворачивает направо
Машина ускоряется
```

Получилось! Подпрограмма right() работает. Такие подпрограммы называются *процедурами*. Конечно, такая конструкция выглядит странно. Зачем нужны подпрограммы, если можно обойтись без них?

Подпрограмма может работать много раз. Давайте создадим две процедуры и будем запускать их в разной последовательности.

```
def right():#поворот направо
    print ('Машина замедляется')
    print ('включается поворотник')
    print ('Машина поворачивает направо')
    print ('Машина ускоряется','\n')

def left():#поворот налево
    print ('Машина замедляется')
    print ('включается поворотник')
    print ('Машина поворачивает налево')
    print ('Машина ускоряется','\n')

right()
left()
right()
left()
```

```
Машина замедляется
включается поворотник
Машина поворачивает направо
Машина ускоряется

Машина замедляется
включается поворотник
Машина поворачивает налево
Машина ускоряется

Машина замедляется
включается поворотник
Машина поворачивает направо
Машина ускоряется

Машина замедляется
включается поворотник
Машина поворачивает налево
```

Машина ускоряется

Теперь программа выглядит гораздо изящнее. Более того, процедуры можно менять. Если мы не захотим пользоваться поворотником при повороте направо, нам надо просто убрать строчку из процедуры right(). После этого, все повороты направо будут проходить без поворотника.

Может возникнуть вопрос — зачем нам процедуры, если у нас уже есть циклы?

1) цикл можно применить, если мы делаем одно и то же много раз **подряд**. А процедуры можно применять вразнобой.

2) может оказаться, что подпрограмма вообще не применится ни одного раза. Такое тоже бывает. Например, вы прошли игру без единого выстрела, хотя вам разрешалось стрелять. Или вы создали деталь на станке с программным управлением, ни разу не перейдя на повышенные обороты, хотя у станка был такой режим.

Цикл нельзя создать «про запас», а подпрограмму можно.

Урок 10. Подпрограммы. Процедуры с параметрами

Проблема разнообразия

Мы уже знаем, как использовать процедуры.

Например, нам нужно просверлить три отверстия. Такие задачи часто нужно решать, программируя станки с программным управлением.

```
def sverlit():
  glubina=20
  print ('есть результат! Отверстие
глубиной',glubina)

sverlit()
sverlit()
sverlit()
```

```
есть результат! Отверстие глубиной 20
есть результат! Отверстие глубиной 20
есть результат! Отверстие глубиной 20
```

Ура! В нашей детали теперь есть три отверстия. Конечно, в реальности всё сложнее. Станок проверит толщину и длину сверла, настроит расположение детали, угол сверления, будет следить за скоростью сверления, покажет нагрев сверла и детали и тд. На самом деле, мы создаём процедуру, чтобы не прописывать все эти предварительные действия каждый раз.

В разных ситуациях, мы применяем разные процедуры. Допустим, нам надо делать мелкие и глубокие отверстия. Для этого очень удобно сделать две разные процедуры:

```
def sverlit_melk():
  glubina=10
  print ('есть результат! Отверстие
глубиной',glubina)

def sverlit_glub():
  glubina=30
  print ('есть результат! Отверстие
глубиной',glubina)

sverlit_melk()
sverlit_glub()
sverlit_melk()
```

```
есть результат! Отверстие глубиной 10
```

```
есть результат! Отверстие глубиной 30
есть результат! Отверстие глубиной 10
```

Бывает, что возможных ситуаций слишком много. Что если от вас могут потребовать отверстие любой длины? 10мм? 11мм? 12Мм? Придумывать отдельную процедуру для каждого из них абсурдно.
Специально для таких случаев, существуют *процедуры с параметром*.

процедуры с параметром

Это особые процедуры, куда можно подставить какое-нибудь значение. Вот как это выглядит:

```
def sverlit(x):
  print ('есть результат! Отверстие
глубиной',x,"\n")

sverlit(10)
sverlit(11)
sverlit(12)
```

```
есть результат! Отверстие глубиной 10
есть результат! Отверстие глубиной 11
есть результат! Отверстие глубиной 12
```

Параметр - это особая переменная (она даже создаётся очень необычным способом). В данном примере у нас есть параметр **x**. Переменная-параметр **x** получит значение, когда вы запустите процедуру. Каждый раз мы запускаем одну и ту же процедуру с разным параметром. Это похоже на работу микроволновки. Она умеет только разогревать, но в неё можно положить разные продукты и получить разные результаты.
Положил сосиску - получил горячую сосиску
Положил пиццу - получил горячую пиццу
Положил котлету - получил горячую котлету

Процедура с параметром - замечательный инструмент. Она может автоматизировать вычисления и сэкономить кучу кода. Например, одна процедура может подсчитать цену товара:

```
def snickers_cost(x): # x - это количество
сникерсов
    cost=40
    print('количество сникерсов:',x)
    print("с вас",x*cost,"руб")

snickers_cost(1) # один сникерс
snickers_cost(2) # два сникерса
snickers_cost(3) # три сникерса
```

```
количество сникерсов: 1
с вас 40 руб
количество сникерсов: 2
с вас 80 руб
количество сникерсов: 3
с вас 120 руб
```

Конечно, переменная не обязательно должна называться x. Лучше всего будет дать ей «говорящее» имя, например, snickers_cost(kolichestvo) или snickers_cost(count).

В процедурах с параметрами, надо внимательно следить за типом данных. Посмотрите, что получится, если я укажу «2» вместо 2:

```
def snickers_cost(kolvo):
    cost=40
    print('количество сникерсов:',kolvo)
    print("с вас",kolvo*cost,"руб")

snickers_cost("2")
```

```
количество сникерсов: 2
с вас 2222222222222222222222222222222222222222 руб
```

Мы попробовали умножить число 40 на символ «2». Python послушно сделал это. В его понимании, он должен был показать символ 40 раз.

Если вы пишете процедуру для другого программиста, есть смысл написать коротенькую инструкцию по применению.
Напишите, что делает ваша процедура, что она принимает в качестве параметра и что она выдаёт.
Это облегчит жизнь вашим коллегам.

Азы локальных переменных

Поговорим о процедурной переменной - параметре. Это особая переменная, живущая только внутри процедуры. Попробуем посмотреть на неё снаружи.

```
def snickers_cost(x):
    cost=40
    print('количество сникерсов:',x)
    print("с вас",x*cost,"руб")

snickers_cost("2")
print(x)
```

```
NameError: name 'x' is not defined
```

Python жалуется, что не может найти переменную x.
Переменная-параметр **x** живет только внутри процедуры snickers_cost(x). Ни до, ни после самой процедуры, эта переменная не обнаруживается. Снаружи она не видна, не существует. Python пытается найти **x** снаружи процедуры и не находит.

Если вы пришли в банк, взяли талончик и дождались очереди, вы можете предъявить талончик сотруднику и рассказать о своей проблеме. *Внутри банка талончик работает.* Но, если вы выйдете с этом талончиком на улицу и предъявите прохожим, они вам ничем не помогут. *Снаружи банка талончик не работает.*

Такие переменные называют локальными. Про них говорят, что они существуют только в пределах процедуры.

Когда несколько человек работают над одной программой, процедуры пишутся разными людьми. Часто возникает вопрос: «не создавал ли кто-нибудь такую же переменную в другой процедуре?» Локальные процедуры помогают избежать подобных проблем.

```
def mars_cost(x):
    print("с вас",x*50,"руб")

def snickers_cost(x):
    print("с вас",x*40,"руб")

def bounty_cost(x):
    print("с вас",x*45,"руб")

mars_cost(4)
snickers_cost(3)
bounty_cost(2)
```

```
с вас 200 руб
с вас 120 руб
с вас 90 руб
```

Здесь используется три разных переменных x в трёх разных процедурах. Как видно, они отлично уживаются вместе и не конфликтуют.

Процедуры с несколькими параметрами

Можно ли создать процедуру с несколькими параметрами? Конечно! Такие процедуры ещё мощнее и ещё полезнее.

Вот пример такой процедуры. Давайте укажем не только глубину сверления, но и скорость (это важно, чтобы сверло не перегревалось при работе с твердыми сплавами металлов):

```
def sverlit(depth,speed): #depth-глубина,speed-
скорость
```

```
  print ('Готово! Отверстие глубиной',depth, ",
скорость",speed)

sverlit(15,6)
sverlit(25,5)
```

```
Готово! Отверстие глубиной 15 , скорость 6
Готово! Отверстие глубиной 25 , скорость 5
```

Процедуры с несколькими параметрами идеально подходят для математических формул. Они используются где угодно, от расчетов астрофизиков до создания компьютерных игр.

Вычислим площадь прямоугольника:

```
def surface(a,b): # площадь (англ. surface)
  print ('Площадь прямоугольника',a*b)

surface(7,5)
surface(8,8)
```

```
Площадь прямоугольника 35
Площадь прямоугольника 64
```

Рассчитаем атаку в компьютерной игре. Допустим, наш герой бьёт монстра волшебным мечом по голове. Атака зависит от силы игрока, его оружия, брони монстра.

```
# strength - сила игрока, weapon - сила оружия,
enemy_armor - броня монстра
def attack(weapon, strength, enemy_armor):
    print ('Атака',enemy_armor-weapon*strength)

attack(7,5,100)
attack(8,8,150)
```

```
Атака 65
Атака 86
```

Урок 11. Подпрограммы. Функции и возврат значения

Проблема влияния на внешний мир.

Процедура - мощный инструмент, когда дело касается вывода информации.
Пришло время сохранить наш результат. Как мы знаем, есть только один способ сохранять данные — это переменные. Напомним, переменные — это именованная область памяти, где лежат числа, тексты и другие куски данных.

Давайте посмотрим на пример. Будем покупать «Сникерсы», а потом считать их количество. Тут будет ошибка, так и должно быть, не волнуйтесь и не пытайтесь её разобрать, она сложная. Не надо разбирать текст ошибки, переходите к обсуждению примера.

```
snickers_count=0 #количество сникерсов

def buy_snickers():#процедура покупки сникерсов
    snickers_count=snickers_count+1 #сникерсов прибавилось!

buy_snickers()

print ('snickers_count')
```

```
local variable 'snickers_count' referenced before assignment
```

Позже мы вернемся к тексту ошибки, а сейчас просто разберём программу.

В теории, мы сделали четыре действия:

1. установили изначальное количество сникерсов (*ноль* штук)
2. создали процедуру buy_snickers(). Она увеличивает количество сникерсов на 1
3. применили процедуру buy_snickers()
4. подсчитали количество сникерсов (их должна быть *одна* штука)

Но это ошибочное мнение. Дело в том, что процедура видит переменную snickers_count, но **не может её изменить**. Вам кажется всё это слишком сложным? Самый простой способ — не пользоваться подпрограммами. Увы, современное программирование слишком сложное, мы вынуждены это делать, иначе код станет слишком громоздким.

Почему же подпрограмма не может поменять переменную snickers_count?
Давайте разбираться.

Глобальные и локальные переменные.

Почему же наша подпрограмма не смогла изменить созданную нами переменную? Так было решено создателями языка Python. **Подпрограммы не имеют полномочий менять переменные, находящиеся снаружи.**

Вот пример, где процедура видит внешнюю переменную:

```
kolichestvo_snikersov=0

def skolko_snickersov(): #создание процедуры
    print (kolichestvo_snikersov)

skolko_snickersov() #запуск процедуры
```

```
0
```

Обратите внимание, мы пишем print(snickers_count) внутри процедуры. Мы глядим на внешнюю переменную как бы «изнутри» процедуры.

Итак, **глобальные (внешние) переменные — это переменные «снаружи» подпрограммы.** Подпрограммы их видят, но не могут их изменить.

Вообще, подпрограмма может создавать и свои переменные. Она может менять их, как пожелает.
Вот пример, где процедура создаёт и меняет переменную внутри себя:

```
def paketi_na_kasse(): #берём пустой пакет на
кассе
    paket=0
    print (paket)
    paket=paket+1
    print (paket)

paketi_na_kasse()
```

```
0
1
```

Обратите внимание, мы пишем print(paket) внутри процедуры. Только глядя изнутри, можно увидеть локальные переменные.

Итак, **локальные (внутренние) переменные — это переменные «внутри» подпрограммы.** Подпрограммы их видят, могут создавать и изменять.

Мы разобрались, как подпрограмма видит и безуспешно пытается поменять глобальные переменные.
Мы разобрались, как подпрограмма создаёт и успешно меняет локальные переменные.
Мы разобрались (на первых уроках), как создавать и менять глобальные переменные.

Но мы ещё не пробовали менять локальные переменные снаружи подпрограммы!
Давайте попробуем:

```
def paketi_na_kasse():
    paket=0

paketi_na_kasse()
print(paket)
```

```
name 'paket' is not defined
```

“name 'paket' is not defined” означает «переменная paket не существует». Python не смог даже увидеть локальную переменную снаружи. Если бы мы написали print(paket) внутри процедуры paketi_na_kasse(), всё было бы хорошо. Но наш print(paket) находится снаружи подпрограммы и Python не видит локальную переменную. Ни о каком изменении не может идти речи. Мы уже видели нечто подобное, когда работали с локальной переменной-параметром на прошлом уроке.

Локальная переменная не видна снаружи подпрограммы.

Итак, подведём итоги.

Вид переменной	Видна снаружи	Видна изнутри	Изменяется снаружи	Изменяется изнутри
Глобальная (внешняя)	да	да	да	нет
Локальная (внутренняя)	нет	да	нет	да

По сути, есть две проблемы:

1. Локальные переменные вообще не существуют снаружи подпрограммы.

2. Глобальные переменные нельзя поменять изнутри подпрограммы

Глобальные и локальные переменные. Разбор логики Python на сложном примере.

Теперь мы вооружены знаниями о глобальных и локальных переменных. Это два разных мира, которые существуют параллельно. Это было сделано, чтобы переменные не путались между собой. Кроме того, когда разные процедуры могут влиять на переменную, потом сложно разобраться, кто допустил ошибку.

Вернемся к нашей программе по покупке «Сникерсов».

Давайте ещё раз посмотрим на пример. Будем покупать «Сникерсы», а потом считать их количество.

```
snickers_count=0 #количество сникерсов

def buy_snickers():
    snickers_count=snickers_count+1

buy_snickers()
```

```
print ('snickers_count')
```

```
local variable 'snickers_count' referenced before assignment
```

Как и ожидалось, процедура не может изменить snickers_count. Она даже не пытается это сделать.
На самом деле, наша процедура попыталась найти *локальную* переменную snickers_count. Вот ход её мыслей:

> «Меня попросили изменить переменную snickers_count. Программист знает, что я могу изменять только локальные переменные. Значит, нет смысла даже пытаться искать её снаружи. Поищу переменную snickers_count внутри себя.
>
> ...через некоторое время…
>
> Как жаль, я всё обыскала внутри себя, но ничего не нашла. Не существует локальной переменной snickers_count. Надо сказать программисту, что он пытается менять переменную, которой нет, а потом сказать Питону, чтобы он остановил всю программу, так как возникла неисправимая ошибка. Итак…
>
> *local variable 'snickers_count' referenced before assignment*
>
> *локальная переменная snickers_count упомянута до своего создания*
>
> … Готово. Надёюсь, программист всё поймёт правильно и, в следующий раз, не забудет создать локальную переменную, прежде чем её менять.
> »

Оказывается, ошибка была не в том, что процедура не смогла изменить внешнюю переменную (хотя мы хотели именно этого). Python подумал, что мы хотели изменить какую-то локальную переменную snickers_count, но просто забыли её создать.

Передача управления подпрограмме в старых языках программирования.

При вызове подпрограммы, текущий код становится на паузу и передаёт управление подпрограмме. Подпрограмма работает, а потом возвращает управление в то место, где происходил вызов. В некоторых старых языках программирования, в начале подпрограммы писали take, а в конце подпрограммы — return. Это выглядело так:

take #(принять управление)

… … …

… … …

… … …

return #(вернуть управление)

Язык Python считается современным, тут ничего не пишут :-). Но процессы здесь те же самые.

Диалог между программой и подпрограммой мог бы звучать так:

- держи пульт!

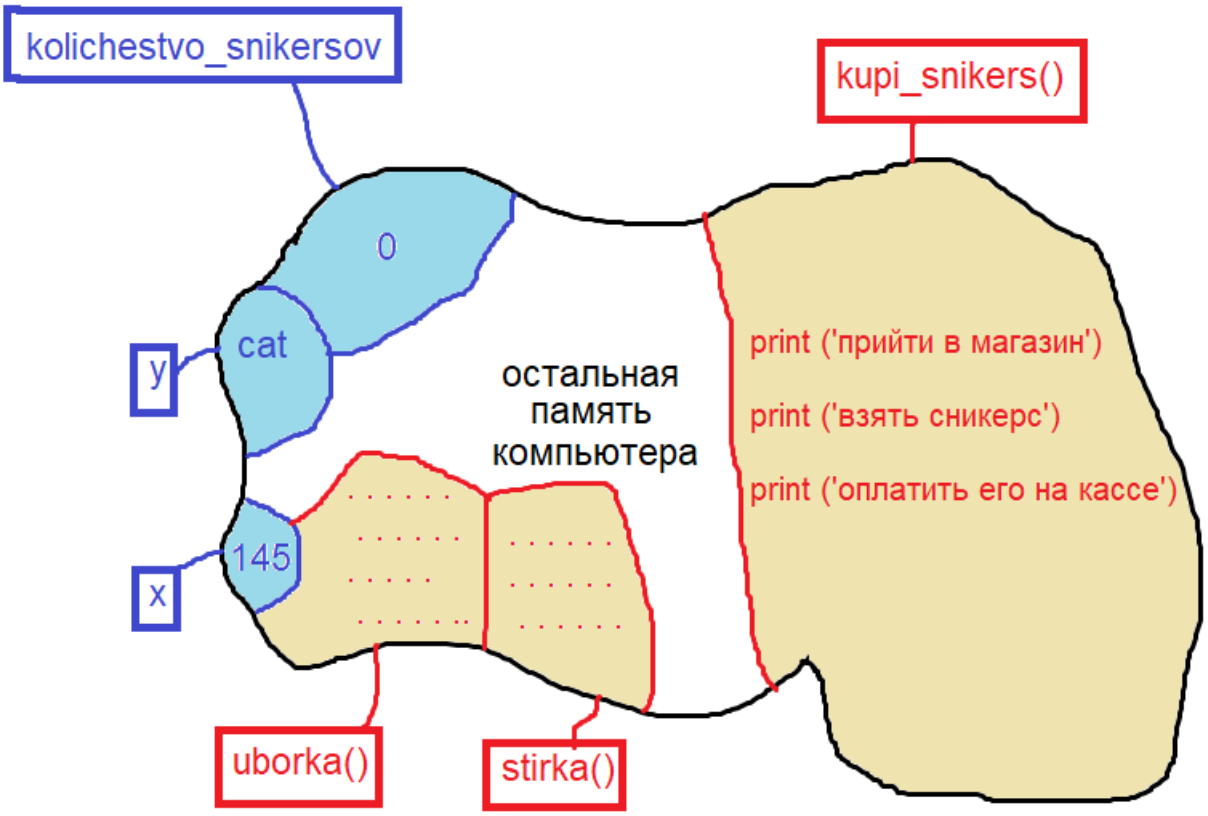
… … …

- Спасибо! Возвращаю!

Память компьютера: данные и инструкции

Память компьютера забита данными и инструкциями. Это переменные и подпрограммы. Они имеют заголовки (имена переменных и подпрограмм).
Если мы хотим запустить подпрограмму — пишем её заголовок в нужном нам месте.
Если мы хотим распечатать переменную — пишем её заголовок внутри print().

На этом рисунке отображены **данные** и **инструкции**.



Мы можем менять переменные. Что можно внести в переменную? Конечно, различные данные. Числа, куски текста и тд.
Но что произойдёт, если мы дадим переменной не данные, а инструкцию?

```
def kupi_snickers():
    print ('прийти в магазин')
    print ('взять сникерс')
    print ('оплатить его на кассе')

kolichestvo_snikersov=0

kolichestvo_snikersov=kupi_snickers()

print("количество
сникерсов:",kolichestvo_snikersov)
```

```
прийти в магазин
взять сникерс
оплатить его на кассе
количество сникерсов: None
```

Когда Python увидел kolichestvo_snikersov=kupi_snickers(), он послушно пошёл в подпрограмму kupi_snickers() и начал искать там данные. Попутно (раз уж он уже тут) Python выполнил все команды этой подпрограммы.
Увы, данных там не оказалось. Это неудивительно, это же подпрограмма, а не переменная.

Специально для таких случаев, Python имеет специальный тип данных None. Это не число и не текст. Это «пустота», «ничто», заглушка для подобных досадных ситуация.

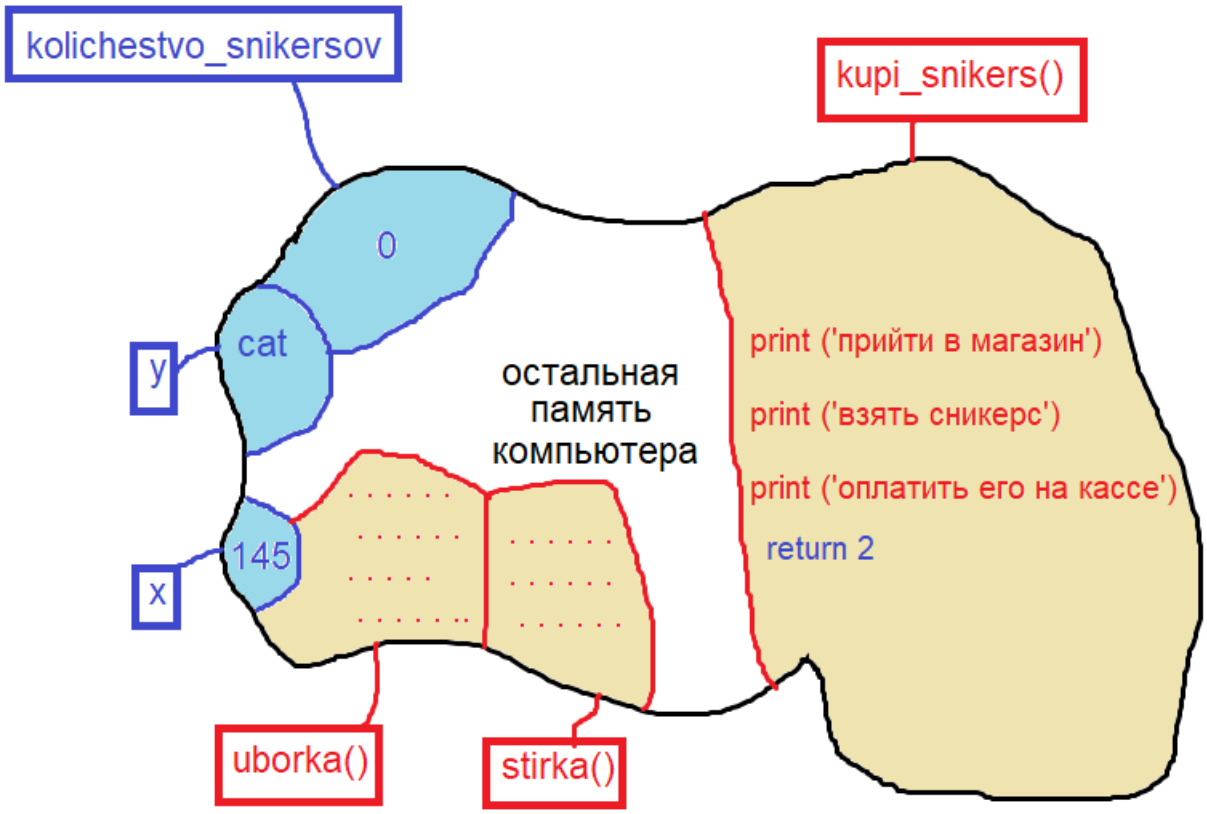
Создание функций

Создатели языков программирования, всё же, очень хотели, чтобы в переменную можно было запихнуть не только числа, тексты, другие переменные, но и подпрограммы.

Специально для этого, были созданы особые подпрограммы, «**функции**».

Функция — это подпрограмма, где, кроме инструкций, есть данные. Это особый гибрид подпрограммы и переменной.

На этом рисунке показана именно такая функция kupi_snickers(). Она имеет команды, как любая подпрограммы. **Но, при возврате управления, она передаст число 2.**



Применим её!

```
def kupi_snickers():
    print ('прийти в магазин')
    print ('взять сникерс')
    print ('оплатить его на кассе')
    return 2

kolichestvo_snikersov=0

kolichestvo_snikersov=kupi_snickers()

print("количество
сникерсов:",kolichestvo_snikersov)
```

```
прийти в магазин
взять сникерс
оплатить его на кассе
количество сникерсов: 2
```

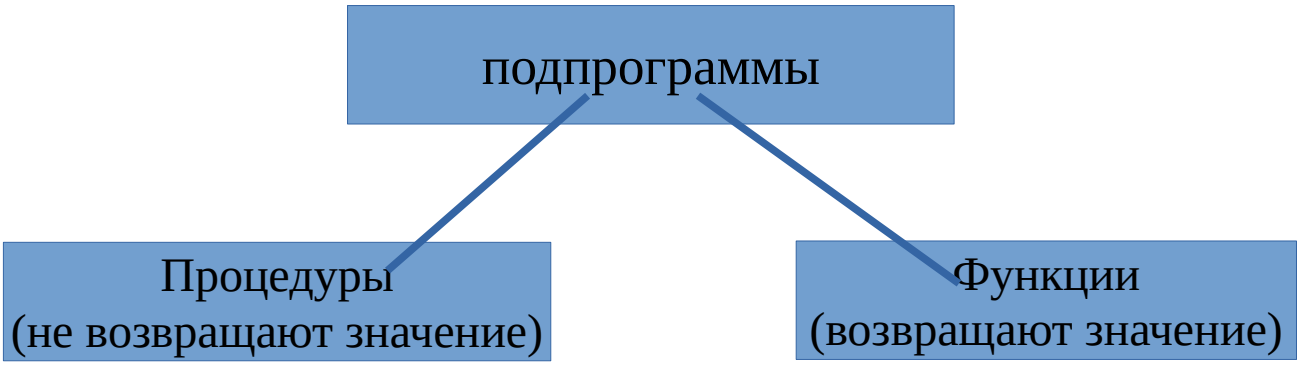
Создатели Python решили использовать в данной ситуации архаичное и устаревшее слово **return**.

На первый взгляд, функции бессмысленны. Зачем эти сложности? Можно было просто написать **print(2)**.
Но это единственный способ сохранить данные из подпрограммы во внешней переменной.

Конечно, в таких простых случаях, как наш, функции не используют. Однако, в больших программах без них не обойтись.

Функция — это подпрограмма, которая имеет **выходное значение.**

Этот процесс принято называть **возврат значения**.



Видоизменённый диалог:

- держи пульт!

- Спасибо! Возвращаю! Кстати, вот тебе маленький подарок, число 2!

Функции позволяют сохранить во внешней переменной результаты вычислений:

```
def surface(a,b): # площадь (англ. surface)
    print ('Площадь прямоугольника',a*b)
    return a*b

s1=surface(7,5)
s2=surface(8,8)
```

```
Площадь прямоугольника 35
Площадь прямоугольника 64
```

Помните, return – это последнее действие функции, это передача управления обратно главной программе. После return функция закроется навсегда. Некоторые программисты говорят, что return – это самоубийство функции.

Всё!

На этом, наш курс программирования подошёл к концу.

Вы уже не новички! Вы разбираетесь в переменных, знаете основные типы данных, умеете считать и строить логические конструкции, работаете с циклами.

Теоретически, этого достаточно для программирования. Но ваши знания на этом не заканчиваются. Вы умеете пользоваться списками, создавать процедуры, знаете о глобальных и локальных переменных и даже применяете функции.

Конечно, программирование не исчерпывается этим набором. Вам предстоит познакомиться с объектами и классами, научиться создавать структуры данных, работать с компонентами и модулями…

Но главный шаг вы уже сделали. Вы овладели азами программирования. На этом автор прощается с вами.

До новых встреч! ^_^

Оглавление