



E-kursuse

# **Teeme ise arvutimänge – algus**

materjalid

Aine maht 3 EAP

**Ljubov Feklistova, Lidia Feklistova (Tartu Ülikool), 2013**

## Содержание

НЕДЕЛЯ 1: ВСТУПЛЕНИЕ И ОСНОВНЫЕ ПОНЯТИЯ.....	5
Предисловие .....	5
Приступим!.....	6
Первые пробы.....	7
Первая программа .....	9
Сообщения об ошибках .....	10
Первая игра.....	11
Основные понятия .....	12
Переменные .....	13
Числа и текст .....	15
Математика.....	17
Основные действия .....	17
Увеличение и уменьшение.....	20
Очень большие и маленькие значения.....	20
Модуль math .....	21
Типы данных .....	21
Ввод и вывод .....	23
Приказ "print" , end="", запятая и плюс .....	24
Ввод данных из файла или веб-страницы .....	25
Что ты узнал? .....	26
НЕДЕЛЯ 2: РЕШЕНИЯ, РАЗВЕТВЛЕНИЕ И ЦИКЛЫ.....	28
Принятие решений.....	28
Проверка условий .....	29
Проверка условий I.....	31
Проверка условий II.....	32
Циклы.....	34
Циклы FOR .....	34
Бесконечный цикл.....	35
RANGE() .....	36
Переменные цикла .....	37
Цикл с литералами .....	37
Секундомер.....	38
Цикл WHILE.....	38
Выход из цикла .....	39
Комментарии .....	40
Еще раз об игре Угадай Число.....	41

Как разработать игру? .....	41
Что ты узнал? .....	43
НЕДЕЛЯ 3: ЦИКЛЫ ВНУТРИ ЦИКЛОВ И СПИСКИ.....	44
Змейка .....	44
Версия 2 .....	45
Версия 3 .....	46
Двойной цикл .....	48
Примеры .....	49
Многократные циклы .....	50
Комбинации и пермутации .....	50
Пример .....	52
Список.....	52
Как достать элемент списка? .....	54
Изменить элемент списка.....	55
Добавление элемента в список .....	55
Удаление элемента из списка .....	56
Поиск элемента .....	56
Сортировка списка.....	57
Что ты узнал? .....	58
НЕДЕЛЯ 4: ФУНКЦИИ И ОБЪЕКТЫ .....	59
И снова игра Змейка .....	59
Версия 5 .....	59
Версия 6 .....	59
Функции.....	59
Вызов функции.....	61
Аргументы функции .....	63
Возвращение результатов работы функции .....	64
Локальные и глобальные переменные .....	65
Объекты .....	66
Создание объекта .....	68
Установка исходного состояния объекта .....	69
__str__().....	70
self.....	72
Чем объекты хороши? .....	73
Пример .....	74
Что ты изучил?.....	74
НЕДЕЛЯ 5: ГРАФИКА И АНИМАЦИЯ .....	76

Графика.....	76
Создание окна.....	76
Рисуем внутри окна .....	77
Координаты окна .....	79
Цвета .....	79
Расположение различных фигур .....	80
Рамки для фигур.....	81
Произвольная линия .....	82
Текст.....	84
Картинки.....	84
Анимация.....	85
Перемещение по прямой .....	86
Перемещение по кривой.....	87
Управление с клавиатуры .....	88
События, которые можно контролировать.....	89
Управление мышкой.....	90
Финал игры Змейка.....	91
Версия 8 .....	92
Версия 9 .....	92
Что ты узнал? .....	92
НЕДЕЛЯ 6: ЗВУК И ЕЩЁ КОЕ-ЧТО ПОЛЕЗНОЕ .....	93
Звук.....	93
Звуковые файлы .....	93
Rugame.mixer .....	93
Контроль за громкостью звука .....	95
Фоновая музыка .....	95
Кое-что полезное.....	96
Таблица первенства - запись в файл .....	96
Узнать имя игрока.....	97
Итоговая игра Змейка .....	99
Немного для саморазвития .....	99
ИСПОЛЬЗОВАННЫЙ МАТЕРИАЛ.....	100



## НЕДЕЛЯ 1: ВСТУПЛЕНИЕ И ОСНОВНЫЕ ПОНЯТИЯ

### Предисловие

Обычно предисловия это те страницы, которые читатель быстро перелистывает. Конечно же, и в этот раз можешь так сделать и приступить сразу "к делу", однако, никогда не знаешь, без чего ты можешь остаться... Так как следующий текст не очень большой, все-таки взгляни на него!

Материал всего курса состоит из **шести** отдельных книг, по одной книге в неделю. Для последней недели нового материала не предусмотрено. В общих чертах, в книгах рассмотрены следующие темы:

1. **книга:** предисловие, первое знакомство с Питоном, ввод данных, вывод, переменные и т.п.
2. **книга:** разветвление и принятие решений, циклы, задумка своей игры
3. **книга:** циклы в циклах и листы
4. **книга:** функции, объекты и змейка
5. **книга:** графика и анимация
6. **книга:** звук и еще кое-что интересное

**В материалах книг много разных икон и картинок, которые обозначают следующее:**



- приведенный пример обязательно сделай самостоятельно на своем компьютере при помощи Питона;



- видеоматериал (на эстонском языке) обязательный для просмотра; для этого надо нажать на картинку, после чего в отдельном окне откроется видеоролик. Видеоролики сохранены в формате *mp4*. В случае необходимости файлы можно при помощи специальных программ пересохранить в других известных форматах (например, *AVI*) – обязательно

сообщи мне о возникших проблемах.

## Как извлечь максимум пользы из курса?

- Будь активен и не жди готовых решений. Если что-то непонятно или возникают вопросы, обязательно задай их в форуме или руководителям курса. Автор всегда видит свой текст простым и понятным – это психологический фактор.
- При изучении программирования, примерно 95% проблем и вопросов разрешаются при помощи Интернета. Объем данного курса относительно мал, для того чтобы объяснить хотя бы десятую часть философии программирования. Задача данного курса – сделать введение в основы программирования на языке Питон, чтобы ты смог самостоятельно продолжить изучать и развивать приобретенные навыки и умения. **Очень важно**, чтобы во время курса ты научился активно и целенаправленно использовать помощь Интернета.

Ответь на вопросы, находящиеся под ссылкой "Обратная связь", чтобы я знала о твоей успехах и проблемах.

## Приступим!

Лиха беда начало! Поэтому не будем мешкаться!

Прежде всего, для того чтобы начать разговор о программировании игры, надо:

1. установить на свой компьютер Питон (бесплатно)
2. уяснить азы программирования - на этой неделе узнаем, что такое переменные, типы данных, ввод, вывод, как и почему надо делать вычисления - одним словом, будем заниматься подготовкой и накоплением знаний для написания своей игры.



### Установка!

Установить Питон очень легко. Зайди на страницу <http://python.org/download/> и выбери в соответствии с операционной системой твоего компьютера последнюю версию Питона (т.е. *Python 3.3.2*).

Если нашел и нажал на правильную ссылку, то дальше следуй сообщениям, которые будут появляться в диалоговом окне.



Здесь можно просмотреть короткий видеотривок про инсталляцию.

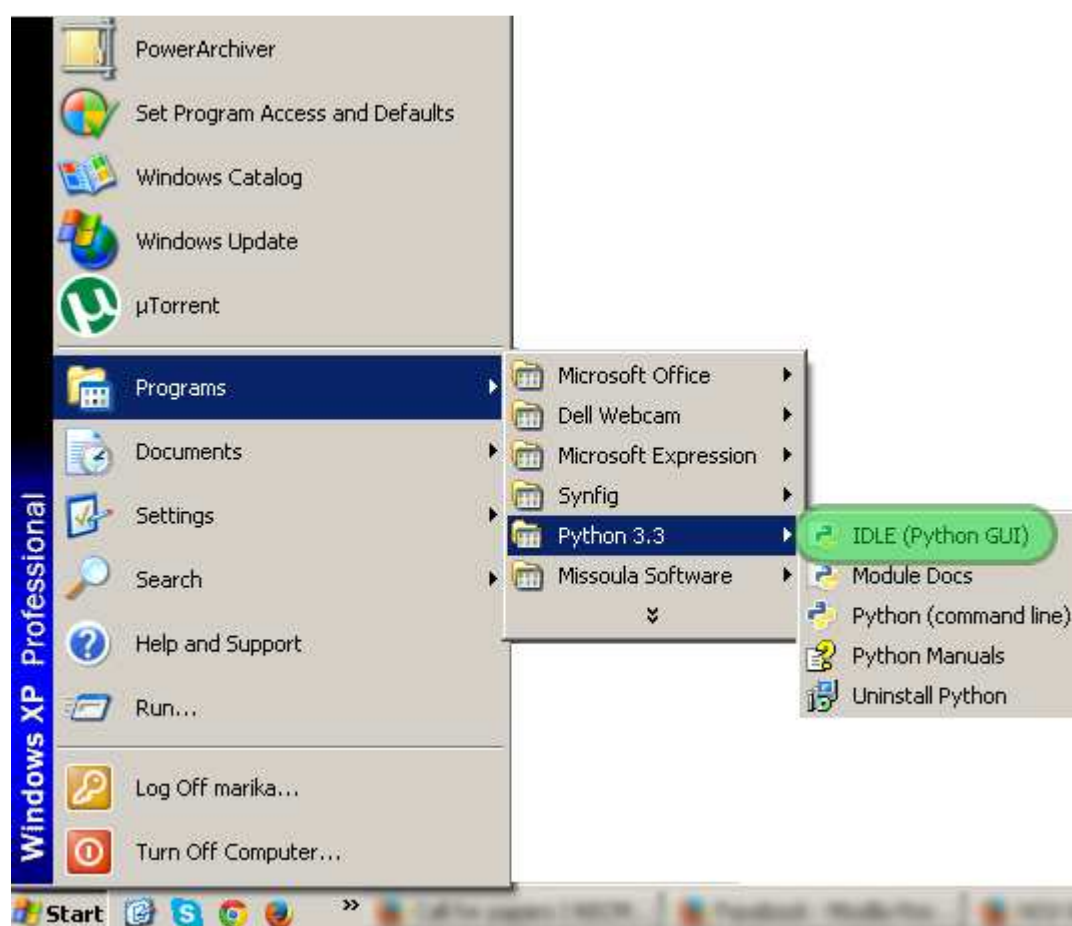
## Первые пробы

Итак, у тебя уже должна быть установлена среда для разработке приложений на языке программирования Питон. Первые пробы можно сделать в консоле **Idle** (англ. *Integrated DeveLopment Environment*) и текстовом редакторе *Idle*. Сперва воспользуемся последним вариантом.

**NB!** Так как я использую операционную систему *Windows*, то много указаний будут касаться именно этой системы. Если у тебя другая операционная система, то следуй по аналогии. При необходимости или сложностях, пошли сообщение в форум.

### Начнем!

Отрой через меню *Старт (Start)*, **Python3.3.2** и выбери **Idle (Python GUI)**. Откроется окно, в котором курсор стоит за тремя угловыми скобочками "**>>>**", называемыми **командной строкой**. Если это строка есть, значит, Питон готов к работе и ждет от тебя команд и инструкций.



## Дай приказ!



Напиши на командной строке после символа >>>:

```
print("Здравствуй, Мир!")
```

Нажми на клавишу ввода (Enter) и напиши:

```
print(Здравствуй, Мир!)
```

Нажми на клавишу ввода и проанализируй разницу между двумя вводами!

```
Python 3.3.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:03:43) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print('Здравствуй, Мир!')
Здравствуй, Мир!
>>> print(Здравствуй, Мир!)
SyntaxError: invalid syntax
>>> |
```



Обрати внимание! Если ты хочешь, чтобы Питон выполнил твой приказ, напиши приказ и нажми на клавишу ввода. Помни, что Питон очень хитрая змея, и все равно какие слова, написанные на консоле, он не понимает. Как ты заметил на картинке, второй приказ не был выполнен, так как нет кавычек.

Еще одно очень важное примечание касается работы в среде *IDLE*: если приказ введен верно, как например **print**, то он подсвечивается фиолетовым цветом; результат - **синим**; ошибки - **красным**. Не забывай, текст пишется в **кавычках**; таким образом Питон понимает, что имеет дело не с приказом, а с текстом и подсвечивает его **зеленым** цветом.

## Попробуем еще!

Напиши на командной строке **print(3+4)**, или **print(6\*4)**, или **print(15/3)**. Что делает Питон?

Попробуй еще раз этот пример, только добавь внутри скобок кавычки!

## Скучно???

Тогда напиши на командной строке и проанализируй следующие строки:

```
print("tee"+"rull"+"uisud")
```

или

```
print("ahv "*20)
```



## Первая программа

До сих пор мы пробовали общаться с Питоном при помощи командной строки, однако программы обычно так не пишутся. **Хотя**, можно и так, если у тебя много времени и не желаешь сохранять проделанную работу.



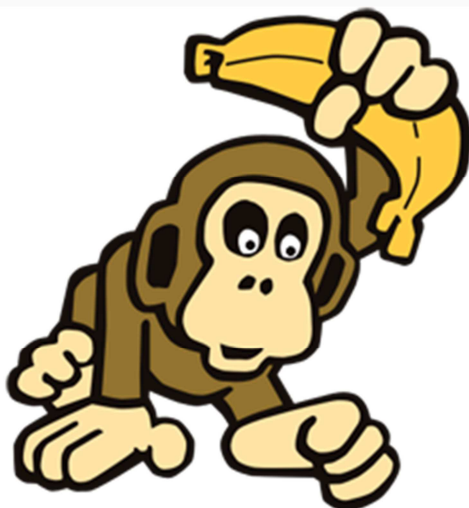
На самом деле, если разрабатывать достойную программу, то для этого придется тебе через меню окна *Idle* открыть *File > New Window*. Откроется абсолютно пустое окно, куда сможешь записать приказ один за другим. Это окно или среду называют текстовым редактором **Idle** или просто **текстовым редактором**. Приказы, которые записываются в редакторе, сохраняются в отдельном файле. Этот файл, т.е. программу, можно запустить в любой момент.

## Как начать?

1. Открой текстовый редактор *Idle*.
2. Сохрани файл, т.е. пустое окно (не обращай внимания, что в файле нет ни одной строчки кода).

- советую создать для пробных программ отдельную папку
- назови программу например *esimene.py*
- не забудь о расширении *.py*

3. В качестве программы напиши следующее:



```
print("Mulle meeldivad ahvid!")
```

```
print("ahvid "*20)
```

```
print("huuh "*50)
```

```
print("Aitab! Kalad on paremad!")
```

4. Сохрани файл заново (**CTRL+s**)

5. Запусти программу: нажми **F5** или запусти через стартовое меню **Run > Run Module**

## Сообщения об ошибках

Что делать, если мною написанная программа не работает, а на экране выводится сообщение об ошибке?

Говоря обобщенно, все ошибки программирования на Питоне можно поделить на две категории:

1. *Syntax errors* - это такие ошибки, которые выявляются еще до начала работы самой программы. Ошибки находит *Idle*, который проверяет твой код, правильно ли написана программа, все ли приказы понятны, все ли пробелы, скобки, кавычки, двоеточия и т.д. находятся в правильном месте. Если нет, то *Idle* выводит сообщение о синтаксической ошибке (т.н. грамматическая ошибка или ошибка написания) и указание, где она примерно находится.

2. *Runtime errors* - это такие ошибки, которые выявляются во время работы самой программы. Ниже приведен пример.

```
print("Tere" + 5)
```

Код записан грамматически верно, но Питон подобных действий выполнять не умеет и выдает сообщение об ошибке.



Подумай и сравни, почему Питон может справиться с приказом `print("Tere" * 5)`, а приказ `print("Tere" + 5)` влечет за собой сообщение об ошибке?

**Ответ!** В общих чертах, этот пример можно сравнить с ситуацией, когда у ученика есть один карандаш и он перемножает его пять раз, т.е. создает пять копий. Однако если у него есть карандаш и добавляю еще 5, возникает вопрос, что он добавляет? Картофелины? Когда к тексту пытаются добавить число, это и есть та самая ситуация с карандашом и картофелинами. Кто может сказать, каков результат подобного сложения: карандаши, картофелины или карандашкартофелины?



**Помни**, в программировании нельзя к тексту добавить число, предварительно не переделав число в текст, или наоборот, текст в число. Об этих превращениях и переводах поговорим чуть позже.

**Помни** так же и о том, что ошибки - это естественный элемент учения. Ошибки допускают даже опытные





программисты. Поэтому настоятельно советую тебе посетить страницу Питона и прочитать дополнительный материал об ошибки и их интерпретации: <http://docs.python.org/dev/tutorial/errors.html>

## Первая игра

### Угадай, какое число я загадал?

Первая программа, которую ты написал на Питоне, ничего особенного не делала - выводила на экран при помощи приказа **print** несколько строчек текста. Так как данный курс рекламировался как курс по созданию компьютерной игры, то давай попробуем создать первую игру.

Во-первых, **будь готов** к веренице кода! Компьютерная игра - это не пара строчек кода, а килобайты и мегабайты кода. Маленькая простенькая игра без графической поддержки занимает примерно 50 строчек кода. Игры, которые будут разрабатываться в данном курсе, занимают примерно 50-200 строк кода.

Так как ты пока еще никаких особых приказов Питона не знаешь, то тебе будет очень сложно создать игру с нуля. Поэтому методика данного курса заключается в обучении программированию и написанию компьютерных игр через примеры и отдельные кусочки кода.



Начнем с самого простого и маленького - игры *Угадай Число*.

1. Открой среду *Idle* и новое пустое окно редактора (**File > New**)
2. Сохрани пустой файл например под именем *ArvaArvu.py*
3. Перепиши нижеприведенный код в файл.

**Зачем перенабивать код, если его можно скопировать?** Дам тебе один простой совет, как быстро и эффективно усвоить языки программирования. Просто копируя программу, ты выполняешь механическую работу и не задумываешься над содержанием строк, для чего ставится двоеточие или отступ. Перепечатывая программу, ты задумываешься над логикой программы и невольно запоминаешь синтаксис.

1. Очень важно в коде следить за отступами, каждый отступ может быть, например, 4 пробела.
2. Обязательно периодически сохраняй программу.
3. Запусти программу клавишей *F5* или **Run > Run Module**.

Удачи!

```

import random
chislo = random.randint(1,999)
print ("Здравствуй! Как тебя зовут?")
imja = input()
print (imja + ", угадай, какое число меньше тысячи я задумал?")
dogadka = int(input())
schetchik = 1

#ниже приведен цикл, который работает до тех пор, пока число отличается
#от задуманного или использовано максимальное количество попыток

while dogadka != chislo :
    #Введенное число больше или меньше задуманного
    if chislo > dogadka :
        print ("Маловато! Предложи большее число")
    elif chislo < dogadka :
        print ("Многовато! Предложи меньшее число")
    dogadka = int(input())
    #Увеличить количество попыток на один
    schetchik = schetchik + 1
#Завершить работу программы, если количество попыток превысило лимита
    if schetchik > 10:
        break

if schetchik <= 10 :
    print ("Молодец, " + imja + "! Ты угадал мое число за ", schetchik, " попыток.")
else :
    print ("Десять попыток использованы. Может, попробуешь изменить тактику?")

```

## Основные понятия

### Основные понятия

Теперь, когда ты немного поиграл со своей первой игрой, тебе, наверняка, хочется узнать подробнее, как создать аналогичную программу?

Спешить однако не стоит. Прежде чем начну объяснять значение каждой строчки игры, следует прежде всего разобраться с некоторыми важными понятиями, такие как **компьютерная память, ввод, обработка ввода, вывод и переменная**.

### Ввод, обработка ввода и вывод

В твоей самой первой программе не было ввода; программа только выводила заранее заданный текст. Скучновата, не правда ли?

Во второй программе (*Угадай Число*) были все три компонента, которые сделали программу увлекательнее. В этой игре:

- **ввод** - вводимые числа и имя игрока;
- **обработка ввода** происходила тогда, когда программа проверяла, вводимое число равно задуманному числу или нет;



- **Вывод** - сообщения об успешности отгадывания.

Понятно, что любая хорошая программа нуждается в вводе; но как и куда компьютер сохраняет вводимые данные? Как какой-то Питон может сказать компьютеру, запомни это сообщение, а это не забывай?

Безусловно, ты знаешь, что у компьютера есть память. **Память** можно рассматривать как большое количество ячеек, которые находятся в одном заданном состоянии до тех пор, пока их не поменяют. Состояние означает в двоичной системе информацию, которую сохранили в данную ячейку памяти. Подобную структуру можно наглядно назвать **запоминанием**. Каждой ячейке присваивается состояние в двоичной системе, и ячейка его сохраняет до тех пор, пока не будет установлено новое состояние. Таким образом, всегда можно изменить ячейку памяти (перепрограммировать) или только прочитать из ячейки записанное в нее в двоичном коде информацию без внесения изменений.

Каким образом Питон или любой другой язык программирования изменяет ячейки памяти? И если содержание ячейки изменяется, как язык программирования найдет нужную ячейку заново? В программировании, чтобы компьютер запомнил важную информацию, надо присвоить этой информации свое уникальное **имя**. Как только имя информации дано, Питон понимает, что надо обратиться к ячейке памяти и сохранить в нее информацию (будь то число, слово, предложение, картинка или музыка). Питон прикрепляет к **имени** адрес ячейки памяти, при помощи которого он сможет, используя имя информации, обратиться к информации. Вот так все просто.

Точно также это выглядит в игре *Угадай Число*. Когда программа спрашивает у игрока число, то этому числу (т.е. информации, которую надо запомнить) дается **имя**, которое помогает сохранить число в ячейке памяти. Таковым именем в данной программе служит *dogadka*; информация прикрепляется к имени при помощи знака равно.

```
dogadka = int(input())
```

## Переменные

На предыдущей страницы было сказано, что любой информации, которую надо запомнить, должно быть присвоено имя, или говоря на языке программистов, создана **переменная**, а саму процедуру, когда переменной дается при помощи знака равенства значение, - **присваиванием значения** переменной.



Проделаем небольшое задание по присваиванию значения переменной в среде *Idle (Python Shell)*.

Присвоим переменной некоторое значение. Что это значит? Напишем на командной строке имя (т.е. переменную), при помощи которого Питон сохранит в ячейку памяти за переменной *значение*, стоящее после знака равенства. Если мы хотим повторно использовать значение (эту информацию), то мы ее получим при помощи той же самой переменной (имени), к которой Питон прикрепил адрес ячейки информации. Однако, лучше один раз увидеть, чем сто раз услышать. Поэтому посмотри видео о переменных и их создании:



### NB! Есть ли ограничения при выборе имен переменных?

Говоря вкратце, можно выбрать в качестве имени любую комбинацию букв - почти! Имя может быть любой длины (однако очень длинные имена усложняют читаемость кода), содержать числа и нижнее подчеркивание (\_). Тем не менее есть еще ряд ограничений:

- имена переменных чувствительны к регистру, т.е. большие и маленькие буквы - это разные символы. Например, переменная *кОшка* и *кошка* - это разные переменные;
- имя переменной должно всегда начинаться с буквы или нижнего подчеркивания;
- имя переменной не должно содержать пробелов, переноса и т.д.

Примеры переменных и присвоенных им значений:

- `imja = "Maria"`
- `pervij_otvet = 45`
- `_imja_uchenika = "Vasja"`
- `Moi_schetchik_2 = 0`
- `TvoiVtoroiOtvvet = "tri porosenka"`

### Как думает программист?

Если переменной присвоено какое-либо значение, то это значение сохранено в памяти и эту информацию теперь можно вызвать при помощи имени переменной. В большинстве языках программирования про это действие говорят, что мы храним значение в переменной. В Питоне действия с переменными происходят чуть иначе, чем в большинстве языках программирования (по моему мнению, даже чуть легче). точнее, вместо того чтобы хранить значения в переменных,

значениям даются имена. Некоторые программисты, которые работают на Питоне, утверждают, что в Питоне нет переменных, а есть только имена. Поэтому в дальнейшем я буду использовать в тексте такие слова, как переменная, имя или имя переменной в качестве синонимов.

## Как изменяются переменные?

Разговаривая о переменных может возникнуть вполне оправданный вопрос, почему имя, присвоенное сохраняемой в памяти информации, называют переменной? Термин говорит сам за себя! В программировании очень распространены ситуации, когда значение, стоящее за используемым именем, надо изменить. Отсюда и название - **переменная**. Например, в игре *Угадай Число*: сначала в данной игре переменная *dogadka* получает первое значение (допустим, 56). Если окажется, что этот ответ неверный, то значение 56 бессмысленно. Спрашиваем у игрока новую догадку и присваиваем старой переменной *dogadka* новое значение, будь то 75, которое надо сохранить в памяти. Используем для этого уже знакомое нам имя. Таким образом, мы стираем т.с. предыдущее значение и переменная *dogadka* получает новое значение 75.

## Числа и текст

Думаю, ты заметил на предыдущей странице, что переменным были присвоены значения двух разных типов: числовые и текстовые (**строковые литералы**).

Примеры числовых переменных:

- `pervij= 4`
- `vtoroj= 6`

Примеры строковых литерал:

- `otvet = "Иной раз поиски правильного решения обходятся дороже ошибки. Дао Цзи Бай"`
- `imja = 'Мария'`
- `chislo_tekstom = "23"`

Как Питон понимает, что имеет дело с числом или текстом (`chislo_tekstom = "23"`)? Вся разница в **кавычках** - если вокруг числа стоят кавычки, то имеем дело с текстом и никаких математических действий с этой переменной сделать не получится.

Если ты был достаточно внимателен, то заметил, что во втором примере (`imja = 'Мария'`) использованы иные кавычки. Однако Питону нет существенной разницы, какие из двух типов кавычек ты используешь, главное, чтобы в начале и в конце сыроковых литерал они были одинаковые.

### Особенно длинные тексты:

Если надо использовать очень длинные тексты (например, стихотворение), то в таком случае используется третий тип кавычек - **тройные кавычки** (`""" """`). Такие кавычки разрешают записывать текст на нескольких строчках с использованием символа *новая строка*.

Пример:

```
Python 3.3.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:03:43) [MSC
tel)] on win32
Type "copyright", "credits" or "license()" for more informati
>>> dlinnoe_slovo = """Во саду ли, в огороде
У Ивана ослик бродит.
Выбирает, выбирает,
Что сначала съесть - не знает. ...

Слева - свекла, справа - брюква,
Слева - тыква, справа - клюква.
Снизу - свежая трава,
Сверху - сочная ботва.

Закружилась голова,
Кружится в глазах листва.
Ослик глубоко вздохнул...
И без сил на землю лег.
(Народное творчество)"""
>>> print(dlinnoe_slovo)
Во саду ли, в огороде
У Ивана ослик бродит.
Выбирает, выбирает,
Что сначала съесть - не знает. ...

Слева - свекла, справа - брюква,
Слева - тыква, справа - клюква.
Снизу - свежая трава,
Сверху - сочная ботва.

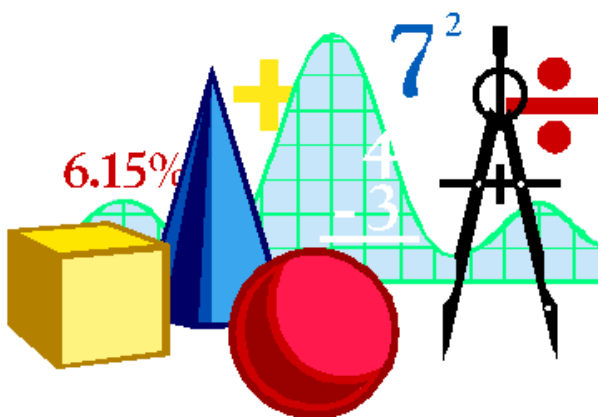
Закружилась голова,
Кружится в глазах листва.
Ослик глубоко вздохнул...
И без сил на землю лег.
(Народное творчество)
>>>
```

Посмотри, какие интересные действия можно сделать с переменными разного типа.



## Математика

Интересно получается, должны были в данном курсе программировать игры, а сейчас заводим разговор о математике. Тебе может показаться банальным, если скажу, что с математикой мы сталкиваемся каждый день, и она играет важную роль в нашей жизни. Но не суди меня



по своим школьным заданиям. Очень часто они не связаны с тем, что нужно в реальной жизни; а если и связаны, то в малой степени. Если ты серьезно хочешь связать свою будущую жизнь с программированием и разработкой компьютерных игр, то в первую очередь, ты должен хорошо освоить курс математики. Почему? Взглянуть на любую компьютерную игру. Она полна объектами разных размеров и с разными физическими свойствами. Объекты движутся на экране по кругу (экран - это ничто иное как система координат) - поэтому программа должна уметь постоянно вычислять скорость движения объектов, направление, месторасположение, цвет (он то же кодируется числами). Подобные программы, полные динамики, создает программист, и если он не силен в математике, то создать сложные конструкции и гениальные идеи будет ему очень сложно.

Поэтому, прежде всего посмотрим, как работает элементарная математика в Питоне. Если усвоишь основные операции над числами на базе одного языка программирования, то сделать то же самое на базе другого языка программирования не составит труда, так как большинство языков программирования очень схожи в этом аспекте.

## Основные действия

Основные действия в Питоне и в большинстве остальных языках программирования делятся при помощи следующих символов:

Символ	Значение и комментарий
+	сложение двух чисел
-	вычитание

*	умножение двух чисел
/	деление
**	возведение в степень
%	нахождение остатка при делении
//	деление двух чисел (результат всегда целое число)
int()	перевод реального числа в целое - убираются цифры после запятой без округления
round()	округление

Прежде чем перейдем к конкретным примерам, посмотрим, чем отличаются **целые числа** (англ. *integer*) и **реальных** (англ. *decimal number*).

Грубо говоря, **целые числа** - это все числа, без запятой ...-3, -2, -1, 0, 1, 2, 3, ... **Реальные числа** - это целые числа, но с запятой + все числа, которые находятся в промежутке между двумя целыми числами ...-2.0,..., -1.0,...,0.0,...,1.0,...,2.0,...

В программирование обычно не используют термин реальное число, вместо этого говорят **число с плавающей точкой** (англ. *floating-point numbers*, или *floats*).

### Примеры с конкретными числами:

Действие	Результат	Комментарий
4 + 5.0	9.0	Сложение целого и реального числа дают в результате реальное число
3 - 5	-2	
3.0 - 5	-2.0	
6 / 3	2.0	Результат деления ВСЕГДА - число с плавающей точкой
5 // 3	1	Деление до целого числа
5 % 3	2	Нахождение остатка

$5 * 3$	15	Умножение
$5 ** 3$	125	Возведение в степень
$4 ** 0.5$	2.0	Нахождение корня числа через возведения в степень
<code>round(2.6375, 2)</code>	2.64	Округление до указанного разряда, <i>здесь</i> , до двух знаков после запятой
<code>round(2.6375)</code>	3	Округление до целого числа
<code>int(2.6375)</code>	2	При переводе текста в число, разряды после запятой удаляются
$3 + 5 * 2$	13	Питон считается с порядком действий; если хочешь изменить порядок действий, используй скобки.
$(3 + 5) * 2$	16	
$6 - 3 - 1$	2	Действия с одинаковыми приоритетами выполняются последовательно слева направо
$6 - (3 - 1)$	4	
$2 ** 3 ** 2$	512	... за искл. возведения в степень; оно выполняется справа налево.
$(2 ** 3) ** 2$	64	



В программирование действует одно железное **правило**: чем больше ты разбираешься и "играешь" с кодом, тем больше и дальше ты преуспеваешь в данной области. Это относится ко всем видеоматериалам и примерам, приведенных в данном курсе. Выполняй самостоятельно все примеры, которые видишь на видео и в тексте, если хочешь преуспеть в программировании, а не в просмотре материала.

### Как считать в Питоне?



Посмотри видеоматериал.



Во многих языках программирования для возведения числа в степень используется символ  $^$ . Ты можешь то же его использовать, но получишь совсем иной результат, чем ты ожидаешь, так как в Питоне он используется в других целях.

## Увеличение и уменьшение

В программировании есть еще два интересных действия, которые в школьной математике обычно не встретишь: увеличение и уменьшение числа на один. Как это работает? Допустим, у меня есть переменная **chislo** = 7. Если надо уменьшить это значение на одну единицу, то это можно упрощенно записать так: **chislo** = -7; а если надо увеличить на одну единицу, то упрощенно можно записать так: **chislo** = +7.

Посмотри видео:



## Очень большие и маленькие значения

Этот раздел добавлен в данную главу скорее для расширения кругозора, чем для обязательного прочтения. Но кто знает, может эта глава тебе понадобится при разработки своей собственной игры.

Обычно новички программирования сталкиваются с очень большими и маленькими значениями, если что-то посчитано совершенно неправильно. Не всегда в этом виноват компьютер.



Попробуй, к примеру, написать на консоле умножение двух очень больших чисел и проанализируй получившийся ответ.

```
>>> 487238562136484.7485*87456874262356.836
4.2612361664542086e+28
>>>
```

## Что буква 'e' и '+' делают в середине числа?

Эта 'e' одна из допустимых записей очень больших и маленьких чисел. Называется она **е-нотацией**. Если расписать результат сложения целиком со всеми разрядами, то это было бы большой головной болью. Во-первых, это взяло бы очень много места, во-вторых, никто не смог бы его правильно произнести. Поэтому в науке часто используется именно е-нотация, которая обозначает следующее:

если записано **4,56E+15** (не важно, использована большая или маленькая *E*), то это означает то же самое, если записать это число так: **4,56x10<sup>15</sup>**. Другими словами, это значит надо передвинуть запятую на 15 разрядов вправо, т.е. увеличить число. Если вместо числа e+15 было



бы записано число  $e-15$ , то запятую надо было бы сместить на 15 разрядов влево, т.е. уменьшить. В результате получим, что

**$4,56e+15 = 4560000000000000$**

**$4,56E-15 = 0,0000000000000456$ .**

## Модуль **math**

Я не стану здесь подробно объяснять, что такое модуль. Для начала ограничимся тем, что модуль можно рассматривать как отдельный пакет программ, которому дано свое имя. В данном разделе речь пойдет о модуле **math**.

Элементарные математические действия можно выполнить в Питоне т.к. "просто так". Следует записать только числа и математический знак. Однако большинство наиболее интересных математических конструкций выполняется при помощи специальных функций, например, синус, косинус, абсолютное значение, логарифм и т.д. Все эти функции и многие другие имеются и в Питоне, однако для их использования в начало программы надо записать несколько ключевых слов:

**from math import \***

**Звездочка** означает, что в программу импортируются ВСЕ функции и переменные модуля **math**. Вместо звездочки можно записать через запятую названия конкретных функций, которые будут использованы в данной программе. Например, если надо только число  $\pi$  (3,14...) и синус из всего пакета, то можно записать так:

**from math import pi, sin()**

Обзор всех функций и переменных пакета **math**, можно найти в официальной документации модуля по адресу:

<http://docs.python.org/release/3.0.1/library/math.html>

## Типы данных

В разделе о переменных я говорила, что существуют разные типы данных. На данный момент ты знаком с тремя: целое число (*integer*), число с плавающей точкой (*float*) и строковая литерала (*string*). Для начала этих трех типов данных нам будет достаточно.

В разделе о переменных говорилось так же и о том, как создать переменную того или иного типа. Здесь мы только напомним парами строк:

- `imja = "Мария"` << **Текстовый литерал** (*string*), используется с кавычками
- `schechik = 3` << **Целое число** (*integer*)
- `skorost = 90.25` << **Число с плавающей точкой** (*float*), используется с точкой

## Изменение типа данных

Довольно-таки часто в программировании требуется изменить тип данных: например, из *integer* в *float* или из *string* в *integer* и т.д. Для этого в Питоне есть три разные функции:

- **float()** переводит текстовый литерал (*string*) или целое число (*integer*) в число с плавающей точкой (*float*)
- **int()** переводит число с плавающей точкой (*float*) или текстовый литерал (*string*) в целое число (*integer*)
- **str()** переводит целое число (*integer*) или число с плавающей точкой (*float*) в текстовый литерал (*string*)

**Примечание!** Скобки рядом с названиями говорят о том, что имеем дело не с обычными приказами Питона, а со специально встроенными функциями. **Функция** - это подпрограмма, которую ты можешь вызывать по имени для выполнения задания. Увидеть функцию в коде легко: за ее именем обязательно стоят скобки, в которых можно дополнительно указать какие-нибудь данные (аргументы), с которыми функция будет в дальнейшем оперировать.

**Например,** если надо перевести целое число 24 в число с плавающей точкой, следует вызвать функцию **float()** и написать в скобках 24. В результате получим число 24.0:

```
>>> float(24)
24.0
>>>
```

Так как о переводе типов данных легче посмотреть видео, чем писать об этом страницы текста, то посмотри следующее видеоматериал. **Не забудь повторить примеры самостоятельно!**



## Как узнать, с каким типом данных имеет дело?

Если осталось не понятным, к какому типу данных принадлежит та или иная переменная, то можно этот вопрос задать Питону при помощи функции `type()`. В скобках надо указать имя переменной. Посмотри на пример, приведенный ниже.

```
>>> a=45
>>> b="Privet"
>>> c="3"
>>> e=5.0
>>> type(a)
<class 'int'>
>>> type(b)
<class 'str'>
>>> type(c)
<class 'str'>
>>> type(e)
<class 'float'>
>>>
```

## Возможные сообщения об ошибках

Само собой разумеется, когда я попытаюсь переделать текст в число, то Питон выдаст сообщение об ошибке, так как он не умеет переделывать текст в числа. А ты умеешь? Другое дело, когда какое-то число представлено в виде текстовой литералы, например "56". Питону эта задача посильна.

## Ввод и вывод

Работу вывода ты уже видел в одних из первых примеров данной книги. При помощи ключевого слова **print()** и скобок, между которыми записано то, что надо вывести на экран (текст в кавычках, значение переменной и т.д.), Питон отправляет данные на выход.

О вводе мы то же уже немного говорили ранее, но не очень подробно. Рассмотрим новую для тебя функцию. Если требуется получить какую-нибудь информацию от пользователя, из файла, веб-страницы, по надо применить функцию **input()**.

**Например, если спросить у пользователя его имя, то делается это так:**

```
>>> imja = input()
Мария
>>> imja
'Мария'
>>>
```

Сначала создается переменная, в которую будут сохранять информацию, полученную от пользователя. Переменная получает значение от пользователя при помощи функции **input()**. Пока пользователь не введет информацию, программа не будет ничего выполнять. Введи свое имя и после этого введи имя переменной заново. Сохранилась ли твоя информация?



Запомни, что функция **input()** считывает данные с консоли в виде строчного литерала. Поэтому когда в игре *Угадай Число* программа должна была сравнить введенное число с задуманным, мы сначала изменили тип данных переменной *dogadka*.

```
print ("Многовато! Предложи меньшее число")
dogadka = int(input())
#Увеличить количество попыток на один

>>> float("Здравствуй!")
Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    float("Здравствуй!")
ValueError: could not convert string to float: 'Здравствуй!'
>>> |
```

## Приказ "print" , end="", запятая и плюс

О выводе шла речь уже несколько раз в предыдущих главах. Здесь я намериваясь рассказать о приказе **print()**, запятой, новой строке и связи между ними.

Помнишь, когда мы хотели вывести на экран особенно длинный текст так, чтобы это было красиво, использовали тройные кавычки. Если у меня возникнет обратное желание, чтобы несколько текстов выводилось на одной строчке, то надо в скобках приказа **print()** в конец добавить приказ *end=""* (*end=""* удаляет по умолчанию прописанный в функции *print* переход на новую строку. Как? Попробуй сохранить следующий пример в отдельный файл, запустить и посмотреть, каков будет результат. Затем сотри *end* и запусти программу заново.



```
print("Если ты споткнулся и упал, это еще не значит, что ты идешь не туда. ", end="")
print("Дао Цзи Бай")
```

Если хочется вывести на экран при помощи одного приказа *print()* данные разных типов, например, текст, числа и еще раз текст, то между данными разных типов надо поставить запятые.



```
print("Если ты споткнулся и упал, это еще не значит, что ты идешь не туда. ", end="")
avtor = input()
print("Если ты споткнулся и упал, это еще не значит, что ты идешь не туда.",avtor)
```

Заметил, что при таком способе соединения данных, Питон по умолчанию добавляет пробел. Если пробел не уместен, то вместо запятых следует использовать знак +.



```
print("Если ты споткнулся и упал, это еще не значит, что ты идешь не туда. ", end="")
avtor = input()
print("Если ты споткнулся и упал, это еще не значит, что ты идешь не туда."+avtor)
```

## Ввод данных из файла или веб-страницы

Иногда бывают ситуации, когда данные надо получить не от пользователя, а из файла или веб-страницы. Как это сделать?

### Чтение информации из файла:

- Создадим переменную, например, **fail**, и присвоим в кавычках ей имя файла, который хотим прочитать (файл должен находиться в том же каталоге, что и программа. В качестве примера, создай в своем каталоге простой .txt файл). Для открытия файла используй приказ **open()**.
- Считываем содержание файла в переменную при помощи приказа **fail.read()** и
  - выводим на экран.



```
fail = open("moi_fail.txt")
soderzanie = fail.read()
print(soderzanie)
```

### Получение информации из веб-страницы:

- Для считывания информации из веб-страницы нам понадобится дополнительный модуль **import urllib.request**.
- Далее действуем аналогично прошлому примеру: создадим переменную, например, **veeb**, присвоим в кавычках адрес странички и для открытия страницы используем приказ **urllib** из дополнительного модуля.
- Создаем новую переменную, в которую сохраним информацию и
- выведем содержание на экран


```
import urllib.request
veeb = urllib.request.urlopen("http://www.math.ut.ee/~nika/soobwenie.txt")
soderzanie = veeb.read()
print(soderzanie)
```

**Примечание!** Если ты попытаешься запустить эту программу в школе, в конторе или в другом офисе, может случиться так, что программа не сработает по той причине, что в офисе используют *прокси* (программу, которая соединяют компьютер локальной сети с глобальной). В целях безопасности прокси могут быть настроены так, что запрещено напрямую делать соединение с интернетом.

## Что ты узнал?

По моему мнению, очень много!

Посмотрим еще раз этот список:

- 
- установка Питона
  - как запустить *Idle*?
  - как пользоваться консолью?
  - видел, как Питон умеет считать и зачем это надо
  - использовать текстовый редактор *Idle* для написания первых программ
  - как запустить программы, написанные на Питоне?
  - сообщения об ошибках
  - как Питон запоминает при помощи переменных?
  - переменные называют так же именами или именами переменных
  - значения переменных могут быть разных типов данных (числа, текст, музыка, картинки, объекты и т.д.)
  - как заставить Питон выполнять разные математические действия?
  - разница между целыми числами и числами с плавающей точкой?
  - как возводить числа в степень (в чем отличие от других языков программирования)?
  - что такое е-нотация?
  - как делить с остатком?
  - перевод значения переменных из одного типа данных в другой

- как узнать, какого типу данных принадлежит значение переменной?
- как работать с вводом?
- как вывести на одной строке переменные с разными типами данных при помощи запятой или знака +
- как вывести на нескольких строках длинный текст?
- как сделать строчный литерал числом?
- как получить информацию из файла?

## НЕДЕЛЯ 2: РЕШЕНИЯ, РАЗВЕТВЛЕНИЕ И ЦИКЛЫ

### Принятие решений

На прошлой неделе мы разобрались с элементарными понятиями программирования, научили компьютер выводить сообщения на экран и делать так, чтобы компьютер спрашивал бы у пользователя данные, считывал из файла или веб-страницы. Также умеем заставить компьютер делать вычисления за нас и умеем сохранять информацию в память. Однако этого маловато для написания серьезной программы, а тем более игры. Думаю, ты согласишься, что программу или игру на уровне ввода-вывода-переменной можно сравнить с игрой прошлого века, нежели с уровнем *AngryBirds*.

У программы или игры должна быть логика. Сегодня ни одна программа без умения самостоятельно принимать решения даже не котируется как программа. Уровень "интеллигентности" программы и ее способной принимать решения



#### Заставь программу принять решение.

В зависимости от введенных данных программа должна реагировать по-разному. Например:

- если нужный файл в каталоге не найден, то надо вывести об этом сообщение;
- если Маша выиграла игру, то она может перейти на следующий уровень;
- если Ваня победил дракона, то должна прозвучать торжественная мелодия.

Для того чтобы принять решение, нужно проверить **условия**. Например, проверим, убит ли дракон, и только после этого запустим играть соответствующий .wav-файл.

Важно понимать, что **результатом** проверки может быть только одно из двух значений: верно или неверно (англ. *true* или *false*). Действие произошло или нет? Компьютеру не получится просто сказать, исполни победную мелодию. Компьютер сделан из железа и понимает простые вопросы. Например,



- Две переменные равны?
- Одна переменная больше второй?
- Одна переменная меньше другой?



В дополнение, сравнение двух переменных не такое уж легкое дело. Переменные, которые сравниваются, должны быть одного типа данных.

Поэтому программист должен сначала перевести движение всех супергероев и врагов в числа, которые компьютер сможет потом сравнивать и на основе сравнения принимать решение. Поэтому историю про Ваню следует переписать следующим образом: если Ваня убил стрелами дракона, то значение переменной *дракон* должно быть равным нулю; контроллер условий проверяет, *дракон=0*; если да, то запускается аудиофайл с победной музыкой; в противном случае, ничего не происходит.

Принятие решение в зависимости от условий называется **разветвлением**. Программа решает, какое действие совершить на следующем шагу.

## Проверка условий

### Конструкция IF

В Питоне условия проверяются при помощи ключевого слова **IF** (обычно ключевые слова подсвечиваются в Питоне **оранжевым** цветом):

```
if Otvet == verno:
    print("Молодец! Это верный ответ")
    schet += 1
print("Спасибо за игру!")
```

В коде, приведенном на картинке слева, проверяется, ответ верный или нет; если ответ верный, то выполняется блок *if*; в противном случае, на экран выводится фраза "Спасибо за игру!"

Почему блок? **Блок** или, другими словами, **тело оператора условий** - это все строчки кода, которые следуют после оператора *if* и выделяются отступом. На примере слева блок составляют две строчки кода. Отступ обычно составляет 4 пробела, но можно отступить на 2 или 5. Главное чтобы во всем коде было единство отступов - **одинаковое количество пробелов** определяло один отступ; в

противном случае, на экране выведется сообщение об ошибке. В целях удобства можно использовать клавишу табуляции.

В программировании на Питоне очень важным знаком является так же двоеточие в конце предложения *if*. Двоеточие сообщает Питону, что теперь начинается тело оператора *if*; если условие верно, то выполняются все приказы блока *if*; в противном случае, приказы блока *if* пропускаются и Питон идет дальше, к приказам, расположенным после блока *if*.

### Двойной знак равенства?

Неужели, на самом деле следует использовать два знака равно для проверки условий? Да, именно так. Запомни, если надо проверить, равны ли значения двух переменных, то надо использовать два знака равно (`==`). Почему? Ординарный знак равенства уже зарезервирован - он используется для присваивания значений переменным.



Неправильное использование знаков `=` и `==` одна из распространенных ошибок программирования. Будь внимателен!

### Другие возможности проверки

Помимо проверки равенства двух переменных, в *if*-предложениях есть и другие знаки проверки:



оператор сравнения вместе с примером	значение
<code>if chislo1 == chislo2 :</code>	Две переменные равны?
<code>if chislo1 &gt; chislo2 :</code>	<i>chislo1</i> больше, чем <i>chislo2</i> ?
<code>if chislo1 &lt; chislo2 :</code>	<i>chislo1</i> меньше, чем <i>chislo2</i> ?
<code>if chislo1 &gt;= chislo2 :</code>	<i>chislo1</i> больше или равно <i>chislo2</i> ?
<code>if chislo1 &lt;= chislo2 :</code>	<i>chislo1</i> меньше или равно <i>chislo2</i> ?
<code>if chislo1 != chislo2 :</code>	Две переменные не равны, т.е. разные
<code>if 20 &gt; chislo &gt; -5 :</code>	<i>chislo</i> находится в промежутке между -5 и 20
<code>if -5 &lt;= chislo &lt;= 20 :</code>	<i>chislo</i> находится в промежутке или равно -5 и 20

## Проверка условий I

При помощи проверок условий можно сделать программу намного интереснее, так как часто требуется проверить не одно условие, а несколько. Для того чтобы программа умела реагировать на разные события по-разному, используй конструкцию *IF-ELIF-ELSE*.

### Конструкция IF-ELIF-ELSE



Для того чтобы лучше разобраться в том, как работает конструкция, приведу наглядный пример. Любая компьютерная игра состоит из множества подпрограмм, которые выполняют разные задачи. Одной из таких подпрограмм может быть проверка условий.

```
print("""Едет Илья Муромец по полю. Видит камень, на нем надпись:
Налево пойдешь (l) - смерть найдешь,
Направо пойдешь (p) - коня потеряешь,
Прямо пойдешь (o) - счастье найдешь.
Помоги Илье принять решение. Введи свое решение при помощи клавиши l, p или o.""")
reshenie = input()
schet = 0
if reshenie == "l" :
    print ("Ты выбрал налево: смерть найдешь.")
    schet -=1
elif reshenie == "p" :
    print ("Ты выбрал направо: коня потеряешь.")
    schet -=1
elif reshenie == "o" :
    print ("Ты выбрал прямо: счастье найдешь.")
    schet +=1
else :
    print ("Такого варианта не дано.")
print ("Ты набрал", schet, "п.")
```

Для начало пользователю предлагается сделать выбор для продолжения игры: нажать на букву l, p или o. Выбор игрока сохраняется в переменной *reshenie*. Кроме этого, в игре используется еще одна переменная *schet*, которая в начале игры равна нулю. В процессе игры запускается проверка условий.

- Если *reshenie* равно l, выполняется первый блок *if*.



Будь внимателен! При сравнении переменных значения должны быть одного типа данных. В рассматриваемом примере пользователь вводит свое решение при помощи приказа *input()*. Это значение рассматривается как текстовой литерал. Для сравнения введенного значения с заданными не забывай про кавычки.

- Если первое условие было не соблюдено (неверно), то Питон идет дальше, к следующему блоку *ELIF* и проверяет условия, прописанные в этом предложении. Если условие не выполнено, Питон идет дальше, не заходя во внутрь блок. Блоков *ELIF* может быть в программе так много, как этого требует твоя программа. Однако большое количество *ELIF* блоков не самое оптимальное решение. Об альтернативных решения мы поговорим чуть позже.
- В куске кода, представленном на рисунке, особенным блоком является последний. Если все предыдущие блоки были проверены и ни один не был выполнен, то имеет смысл добавить блок *ELSE* без каких-либо условий, т.с. блок по умолчанию, который выполняется, если ни один вышестоящий блок не был выполнен (например, пользователь ввел букву, которая не была предусмотрена разработчиком).
- В примере самым последним приказом будет вывод результата на экран. Как видно на примере, последний приказ *print* не находится внутри блоков *if* или *elif*, а стоит сам по себе, значит, он будет выполнен в любом случае.



## Проверка условий II

Иногда более сложные ситуации требуют нескольких условий внутри одного предложения *IF*. Что это значит?

До сих пор мы проверяли только одно условия внутри предложений *IF* или *ELIF*, например, *reshenie == "p"*. В этом разделе увидим, что внутри предложений *IF* или *ELIF* может быть сразу несколько условий.

### AND и OR и NOT

Опять-таки рассмотрим работу операторов на конкретном примере. Предположим, для игры на компьютере программа должна знать возраст пользователя и класс, в котором он учится.

```

print ("Сколько тебе лет?")
vozrast = int(input())
print ("В каком классе ты учишься?")
klass = int(input())

if vozrast > 18 or klass > 10 :
    print ("Ты староват для этой игры.")
elif vozrast < 10 and klass < 4 :
    print ("Тебе еще рано играть в такие игры.")
else:
    print ("Приступим к игре!")

```

- Обрати внимание, что переменные `vozrast` и `klass` должны быть численными типами данных. Метод `Input()` принимает вводимые данные как строковую литералу, поэтому для сравнения возраста и класса нам понадобится приказ `int()` для перевода текста в число.
- В первом предложении *if* проверяется два условия: если одно из условий выполнено, то будет выполнен блок. В данном примере, если возраст пользователя больше 18 **или** ученик учится хотя бы в 11 классе, то выводится сообщение, что пользователь староват для игры. Это сообщение появится, например, и в том случае, если ученик учится в 10 классе, но ему 19 лет.
- Предложение *Elif* проверяет так же два условия. В отличие от предыдущего примера, оба условия должны быть выполнены, чтобы Питон перешел к приказам, написанным внутри блока. В данном примере проверяется, если пользователь младше 10 лет **И** если он учится в младших классах (т.е. с первого по четвертый), то на экране выведется сообщение, что пользователь слишком маленькой для такой игры и ему надо повзрослеть. Это сообщение не выведется на экран, например, в том случае, если пользователю уже есть 11 лет, а учится в 3 классе (выполнено только одно условие проверки).

В случаях когда первое условия должно быть выполнено, а второе, наоборот, требует значения неверно, можно использовать следующую комбинацию условий: **`chislo>10 and not chislo==20`**.

## Циклы

Циклы - очень важная тема в программировании. Для того чтобы понять их значение и важность, обязательно выполни все примеры и задания самостоятельно!

Писать тысячи раз один и тот же кусок кода (например, вывести на экране сообщение "Привет!") - очень обременительное и скучное занятие. Вместо этого можно попросить компьютер, чтобы он самостоятельно выполнил именно этот кусок кода  $n$  раз.



В целом, циклы можно поделить на типа:

- циклы, которые выполняют определенный код заданное количество раз - циклы **FOR**.
- циклы, которые выполняются до тех пор, пока условие работы цикла верно - циклы **WHILE**.

Рассмотрим их подробнее!

### Циклы FOR

Открой текстовый редактор *Idle* (*File > New*), запиши следующие две строчки кода, сохрани программу (например, под именем *tsykkell.py*) и нажми **F5**.



```
for schechik in [1,2,3,4,5]:  
    print ("Питон!")
```

Заметил, что приказ *print* написан только один раз, а слово "Питон" печатается пять раз?!

Именно в этом и заключается вся мощь циклов:

- во-первых, компьютер может выполнить один и тот же кусок кода очень много раз за очень короткое время;
- во-вторых, циклы помогают избежать многократного переписывания кода, тем самым снижая количество опечаток и количество строк в программе.

## Некоторые пояснения к тому, как работает цикл FOR.

### Понятия:

- *schechik* - это **переменная цикла**. Переменная цикла каждый раз получает новое значение, в зависимости от того, какие значения записаны в скобках. Имя переменной может быть любым.
- ключевое слово **in** указывает переменной, откуда брать значения.
- [1, 2, 3, 4, 5] - **список**. Список состоит из значений, которые принимает переменная, и записывается всегда в квадратных скобках.
- после **двоеточия**, учитывая отступы, записывается тело цикла, т.е. блок. В нем прописываются приказы, которые Питон должен циклично выполнять.

### Как работает цикл?



Цикл *for* выполняется ровно столько раз, сколько значений записано у него в списке. В нашем примере, пять значений, т.е. цикл выполняется пять раз. В нашем примере не играло особой роли, какие именно значения были прописаны в списке (числа, литералы), **важно** то, сколько элементов в списке. Попробуй заменить элементы списка на [1, 1, 1, 1, 1] или ['Маша', 'Саша', 'Даша', 'Дана', 'Лана'].

Попробуй вывести на экран значение счетчика:



```
for schechik in [1,2,2,1,105]:  
    print (schechik)
```

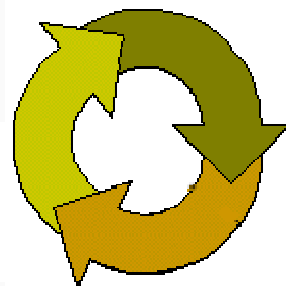


## Бесконечный цикл

**Бесконечный цикл** - это цикл, который работает все время (бесконечно). Обычно такая ситуация случается случайно в ходе какой-то логической ошибки. В случае с циклами *For* ошибки подобного рода случаются редко, в отличие от циклов *While*.

### Что делать с циклом, который работает бесконечно?

Подобный цикл нужно остановить при помощи "силы", т.е. используя комбинацию клавиш **Ctrl+C**.





## RANGE()

Попробуем применить цикл к более реальным заданиям. Например, выведем на экран таблицу умножения.



Открой новое окно *Idle*, напиши в него следующие строки кода, сохрани файл с расширением *.py* и запусти.

```
for schetchik in [1,2,3,4,5,6,7,8,9,10]:
    print (schetchik, "x 7 = ", schetchik*7)
```

Если ты запустил эту программу, то увидел смысл и результат программы. Однако 10 строчек таблицы умножения никакое не чудо. А как заставить компьютер перемножить на семь 1000 чисел? Неужели [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, ... ]? Конечно же, нет! Для таких ситуаций в Питоне есть специальная функция *range()*.

Функция **range()** запускает цикл указанное количество раз минус один. Например, для того чтобы перемножить семь 1000 раз, в скобках следует указать число на одно значение больше - 1001:



```
for schetchik in range (1001):
    print (schetchik, "x 7 = ", schetchik*7)
```

Если требуются значения из какого-нибудь промежутка (например от 20 до 70), то делается это так:



```
for schetchik in range (20,71):
    print (schetchik, "x 7 = ", schetchik*7)
```

В функции *range()* прописываются значения начала и конца работы счетчика. Не забывай, что счетчик заканчивает работу чуть раньше - на одну единицу - чем указано во втором **аргументе** (т.н. число в скобках).

У функции *range()* есть еще несколько интересных свойств: попробуй запустить следующий код:



```
for schetchik in range (1, 10, 2):
    print (schetchik, "x 7 = ", schetchik*7)
```

На этот раз добавили в функцию *range()* третий аргумент. Число, записанное в скобках на третьем месте, говорит функции *range()*, который по счету элемент в списке нужно взять. Так, в примере стоит значение 2, оно говорит функции, что надо брать каждое второе значение.







Попробуй заменить значение 2 на свое число, проанализируй результат.

## Переменные цикла

**Переменные цикла** - это такие же переменные, как ранние изученные переменные, только их используют внутри цикла, а не за его пределом. За пределами цикла переменные цикла не видны и к ним нельзя ссылаться. Имя переменных цикла может быть любым, главное, чтобы оно соответствовало общим принципам Питона. Однако, в программировании есть традиция называть переменные цикла буквами **i, j, k** и т.д. Почему? Потому что на заре программирования ресурсы компьютеров были ограничены и каждая буква была на счету. Имена переменных должны были быть очень короткими.



Если ты решишь применить коротенькие имена переменных цикла к простым переменным, то это укажет на плохой стиль кода. Ниже приведен пример хорошего стиля, соблюдая многолетние традиции программистов:

```
for i in range (1, 10, 2):
    print (i, "x 7 = ", i*7)
```

## Цикл с литералами

В предыдущих примерах мы использовали переменную цикла как счетчик или как число. Теперь посмотрим, что будет, если переменная цикла будет не число, а литерала или последовательность других объектов. Открой в *Idle* окно текстового редактора, запиши код, сохрани и запусти программу.



```
for i in "Добро пожаловать!":
    print (i)
```

Интересно, не так ли? Записали вместо привычной последовательности чисел текст и цикл работает. Получается, Питон рассматривает литералы как последовательность, где каждая буква, символ, пробел - это элемент последовательности. Цикл работает до тех пор, пока в последовательности есть элементы.

Почему Питон записывает каждую букву и символ на отдельную строку? Такова особенность приказа *print()* - переход на новую строку приписан внутри приказа *print()*. Если есть необходимость записать последовательность на одной строке, то последним аргументом в скобках приказа *print()* должен быть **end=""**, который т.с. стирает переход на новую строку.

### Еще один пример:



```
for i in ["Один", "Два", "Три", "Четыре", "Пять"]:
    print (i, " - это числа", end="; ")
```

Запиши, сохрани, запусти и проанализируй самостоятельно.

### Секундомер

В компьютерных играх очень часто приходится выполнять задания на время. Как заставить компьютер считать секунды? Секундомер может работать и без графической поддержки. Как в таком случае увидеть, что он работает правильно? Используем следующий кусок кода, который печатает секунды на экране. Открой текстовый редактор *Idle*, запиши код, сохрани и запусти.



```
import time
for i in range(10,0,-1):
    print(i)
    time.sleep(1)
print("Время вышло!")
```

Разберем



код. Для того чтобы компьютер считал секунды, надо сначала импортировать модуль *time*, затем создать цикл, который считает от 10 до 1 при помощи функции *range()*. В данном примере третий аргумент функции *range()* отрицательный, так как отсчет идет в обратную сторону.

В блоке цикла - два приказа. При каждом заходе в цикл на экран выводится значение переменной цикла и весь процесс приостанавливается ровно на одну секунду (***time.sleep(1)***). После этого проверяется условие выполнения цикла заново: если условие верно, то выполняется блок цикла; в противном случае, Питон переходит к последней строчке нашего кода - *print()*.

### Цикл WHILE

До сих пор мы рассматривали работу только одного типа циклов - *for*. Его особенность заключается в том, что мы всегда знаем, сколько раз будут выполнены приказы тела цикла.

Теперь рассмотрим такие циклы, в которых неизвестно, сколько раз будет выполнено тело цикла. Цикл заканчивает свою работу только тогда, когда условие продолжения цикла нарушено, т.е. не

выполняется. Классический пример использования таких циклов - угадывание чисел.

```
import random
chislo = random.randint(1,999)
print ("Здравствуй! Как тебя зовут?")
imja = input()
print (imja + ", угадай, какое число меньше тысячи я задумал?")
dogadka = int(input())
schetchik = 1

#ниже приведен цикл, который работает до тех пор, пока число отличается
#от задуманного или использовано максимальное количество попыток

while dogadka != chislo :
    #Введенное число больше или меньше задуманного
    if chislo > dogadka :
        print ("Маловато! Предложи большее число")
    elif chislo < dogadka :
        print ("Многовато! Предложи меньшее число")
    dogadka = int(input())
    #Увеличить количество попыток на один
    schetchik = schetchik + 1
#Завершить работу программы, если количество попыток превзошло лимита
    if schetchik > 10:
        break

if schetchik <= 10 :
    print ("Молодец, " + imja + "! Ты угадал мое число за ", schetchik, " попыток.")
else :
    print ("Десять попыток использованы. Может, попробуешь изменить тактику?")
```

## Выход из цикла

Бывают такие ситуации, когда цикл надо завершить на середине работы. Если ты был достаточно внимателен при прочтении материала, то заметил, что в программе *Угадай Число* было показано, как остановить работу цикла. Однако, ты не должен был до конца разобрать суть остановки. Именно в этой главе будет объяснен этот аспект.

### Работу цикла можно остановить двумя способами:

- счетчик цикла растет, а приказы блока не выполняются - приказ **continue**
- полностью завершается работа цикла - приказ **break**

## CONTINUE vs BREAK



Посмотри видео и разбери примеры.



## Комментарии

Во всех программах, кроме игры *Угадай Число*, были записаны только команды для компьютера. Однако очень важно в начале обучения программированию научиться писать комментарии для себя и для тех, кто дальше будет работать с кодом. Посмотрим к примеру игру *Угадай Число*. в ней есть несколько красных строчек, которые компьютер не читает и не выполняет, но в которых содержится очень важная информация или пояснение для разработчиков.

Комментарии очень важны при написании **документации**. Что это значит? Документация - это руководство по использованию программы. В ней можно найти ответы на следующие вопросы:

- Зачем написана данная программа?
- Кто написал эту программу?
- Для кого написана программа?
- Как организована работа программы?
- И т.д. ...

### Как добавить комментарий?

Комментарии можно добавить разными способами.

#### Однострочный комментарий

Каждую строчку можно превратить в комментарий, если поставить в ее начало знак **#**. Все, что находится за этим знаком, подсвечивается красным цветом. Питон понимает, что эту строчку он должен проигнорировать. Однострочные комментарии очень удобны для отключения одной строчки кода, если программа не делает то, что ожидается, или для поиска ошибки.

#### Многострочные комментарии

Если требуется записать более длинные пояснения, то следует использовать многострочные комментарии. Можно использовать знак **#** в начале каждой строки, но удобнее использовать тройные кавычки. Если предпочтешь первый вариант, то есть смысл добавить строчку со звездочками в начале и конце комментария - это выделит комментарий из общего кода. Именно такой стиль используется в начале программы, куда записываются общие данные о программе, программисте, дате и т.д.

```
""  
Это комментарий, подсвеченный зеленым цветом.  
Все, что записано между кавычками,  
программой не читается и не выполняется.  
""
```

```
#*****  
#Это комментарий  
#Он создан в целях демонстрации примера.  
#29.09.2013  
#Любовь Феклистова  
#*****
```

## Еще раз об игре Угадай Число

Теперь, когда основные компоненты игры *Угадай Число* разобраны, посмотрим на программу еще раз целиком. В этом тебе поможет видеофайл.

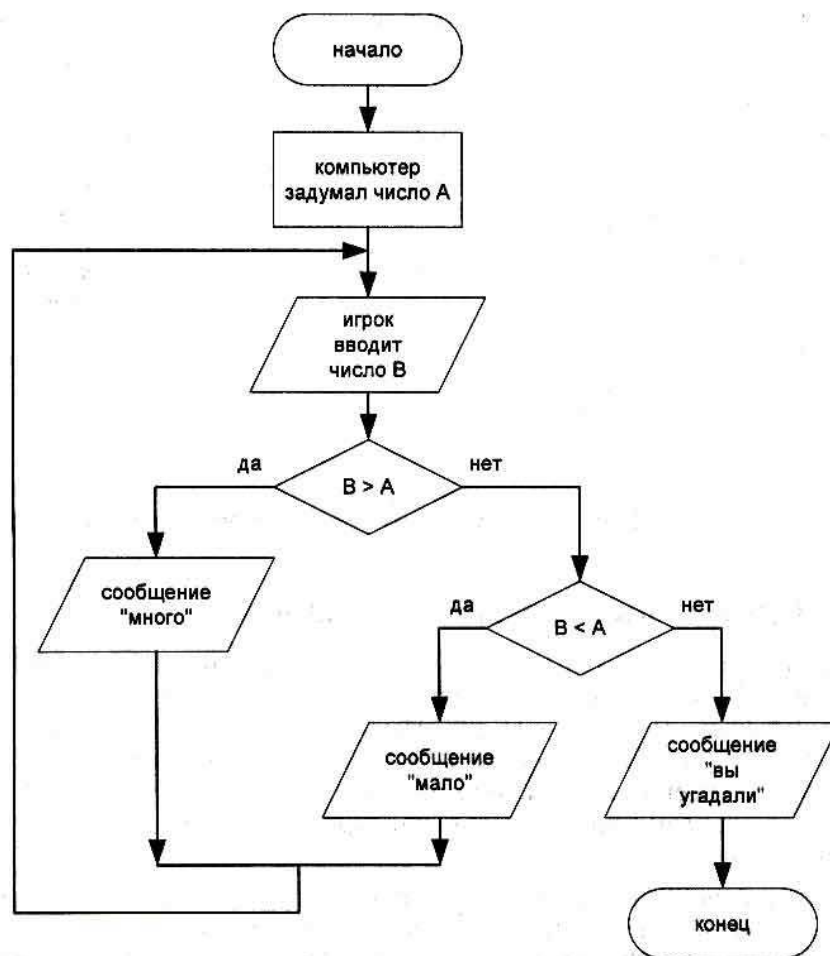


## Как разработать игру?

Прежде чем приступить к программированию игры, следует подробно расписать для себя, в чем будет заключаться суть игры и каковы основные действия в ней. Последнее означает, продумать ход игры поэтапно. Например:

- если пользователь нажмет на стрелку вправо, собачка передвигается на три шага вправо.
- если пользователь вводит число, то проверяется верность ответа и т.д.

Написание алгоритма программы можно сравнить с черчением плана дома. Обычно никто не начинает строительство дома до того, как на бумаге не готов детальный план дома. В программировании точно так же, особенно когда имеем дело с большими программами (большими, чем мы до сих пор рассматривали). Алгоритмы записывают или рисуют при помощи блок-схем, как показано на примере внизу (Источник <http://tat67183862.narod.ru/primer.htm>):



### При создании схемы надо учитывать:

- У алгоритма всегда **одна точка начала** и **одна точка конец**; изображаются **овалами**.
- Нейтральные действия, такие как сообщение, вычисление, увеличение счетчика, движение из одной точки в другую и т. д. изображаются внутри **прямоугольника**.
- Действия, которые требуют принятия решений, т. е. разветвление, изображаются **ромбами**. В ромбах так же изображают циклы, где проверяются условия выполнения (*while*). Из ромба выходят всегда две стрелки, направление **ДА** и **НЕТ**, т.е. ситуации, когда условие действует, и ситуации, когда условие не выполнено. Внутри ромба записывается вопрос.
- В **параллелограммах** изображается обмен данными с пользователем.
- В схеме не должно быть ни одной пустой стрелки или блока без выхода. Схема составлена правильно, если из точки начала можно попасть в точку конца.

## Что ты узнал?

Посмотрим еще раз, какие темы были рассмотрены на этой неделе:

- как заставить компьютер принимать решения?
- как работает конструкция *If-elif-else*?
- как одновременно проверить несколько условий?
- как скомбинировать проверку нескольких условий в одном *if*-предложении при помощи *and*, *or* и *not*?
- что такое цикл?
- как работает цикл *for*?
- что такое список?
- что работает функция *range()*?
- как при помощи функции *range()* задать промежуток?
- функция *range()* по умолчанию начинается с нуля
- функция *range()* заканчивает работу на один цикл раньше, чем указано в скобках
- как в функции *range()* перепрыгивать через несколько значений?
- что делать в случаях, когда цикл работает бесконечно?
- как спроектировать секундомер?
- как и зачем использовать цикл *while*?
- как остановить цикл?
- как и зачем добавлять в программу комментарии?
- комментарии нужны в первую очередь самим программистам (так как через пару недель забудется, что делал тот или иной кусок кода)
- код программы можно закомментировать в целях поиска ошибки
- с чего начинается разработка игры?
- что такое блок-схема?





## НЕДЕЛЯ 3: ЦИКЛЫ ВНУТРИ ЦИКЛОВ И СПИСКИ

### Змейка



Ты уже, наверное, не можешь дождаться, когда перейдем к разработке графических игр. Вот сегодня и перейдем к разработке такой игры, но до самой графики мы доберемся только в начале 5 недели.

В этом курсе мы будем учиться разрабатывать игры через практические задания; часто, это будет подразумевать переделывание и умение ориентироваться в чужом коде. Перепиши приведенные ниже 25 строчек кода в текстовой редактор *Idle*, сохранение его под названием *uss1.py*.

Я представляю, сейчас твое удивленное лицо и возмущение: Что??? Перенабирать 25 строк кода?! Поверь, это один из лучших способов научиться программировать и разбираться в коде. Вникая в суть каждой строчки, ты понимаешь, почему используются те или иные структуры.

Когда ты справишься с набором кода, запусти его. Программа пока ничего интересного не делает, но для вступления сгодится. Попробуй объяснить, что каждая строчка в коде делает. Некоторые вещи будут для тебя новые, но основные конструкции тебе уже знакомы. В данной игре будут использованы приказы из модуля *pygame*. Мы поговорим о них подробнее в пятой книге. Далее мы будем продолжать разрабатывать и дополнять программу, пока из нее не получится цельная и полноценная игра.

И еще! Не забывай сохранять файл время от времени при написании кода! Скопируй картинку Питона себе на компьютере, в тот же самый каталог, где и сам код программы, под названием *pea2.png* (картинка находится справа).



**И самое главное!** Установи себе на компьютер модуль *pygame*. Это маленький сборник программ, который поможет упростить нашу работу с графикой.



### Руководство по установке *pygame*:

- Зайди на страницу Питона: <http://www.pygame.org/news.html>
- Выбери в верхнем левом углу *Downloads*.



- Выбери нужную версию *pygame* в соответствии с версией своей операционной системы и *Python*, нажми на ссылку. Учти, что *Pygame* и *Python* должны быть одинаковой разрядности.
- Установи *Pygame* в тот же каталог, куда и *Python*.



С *pygame* связано одно неудобство: *Idle* и *pygame* не очень хорошо ладят друг с другом. Программы легко написать и запустить, но для того чтобы закрыть игру, надо завершить работу с *pygame*.



И вот он, первый кусок кода игры змейка!

```
from pygame import *
from sys import *

init ()
okno = display.set_mode ([640,480])
foto_golova = image.load ('SnailHead.png')      # 85x96 пикселей
wag_zmeiki = 10
x = 50
y = 50

while True:
    okno.fill([255,255,255])
    okno.blit(foto_golova,[x,y])
    display.flip()
    time.delay(10)
    for e in event.get():
        if e.type == QUIT:
            exit()
        elif e.type == KEYDOWN:
            if e.key == K_UP:
                y = y - wag_zmeiki
            elif e.key == K_DOWN:
                y = y + wag_zmeiki
            elif e.key == K_LEFT:
                x = x - wag_zmeiki
            elif e.key == K_RIGHT:
                x = x + wag_zmeiki
```

## Версия 2

Думаю, если оставить пока приказы графики в стороне, то оставшая часть кода тебе понятна, и голова змейки у тебя двигается по экрану при помощи клавиш.



Поэтому добавим еще один кусок кода. Сравни нижеприведенный код с первой версией кода и допиши недостающие строки. Проанализируй, как и почему змейка ведет себя по-другому.

```

from pygame import *
from sys import *

VLEVO = 1
VVERH = 2
VPRAVO = 3
VNIZ = 4
init ()
okno = display.set_mode ([640,480])
foto_golova = image.load ('SnailHead.png')      # 85x96 пикселей
wag_zmeiki = 10
x = 50
y = 50
dvizenie = VPRAVO

while True:
    okno.fill([255,255,255])
    okno.blit(foto_golova,[x,y])
    display.flip()
    time.delay(10)
    if dvizenie == VVERH:
        y = y - wag_zmeiki
    elif dvizenie == VNIZ:
        y = y + wag_zmeiki
    elif dvizenie == VLEVO:
        x = x - wag_zmeiki
    elif dvizenie == VPRAVO:
        x = x + wag_zmeiki
    if x < 0 :
        x = 0
    if x > 600 :
        x = 600
    if y < 0 :
        y = 0
    if y > 430 :
        y = 430
    for e in event.get() :
        if e.type == QUIT:
            exit()
        elif e.type == KEYDOWN:
            if e.key == K_UP:
                dvizenie = VVERH
            elif e.key == K_DOWN:
                dvizenie = VNIZ
            elif e.key == K_LEFT:
                dvizenie = VLEVO
            elif e.key == K_RIGHT:
                dvizenie = VPRAVO

```

## Версия 3

И еще немного о графике. Интересно, что этот код делает?

Сохрани себе на компьютер картинку  под именем *Bang.png*

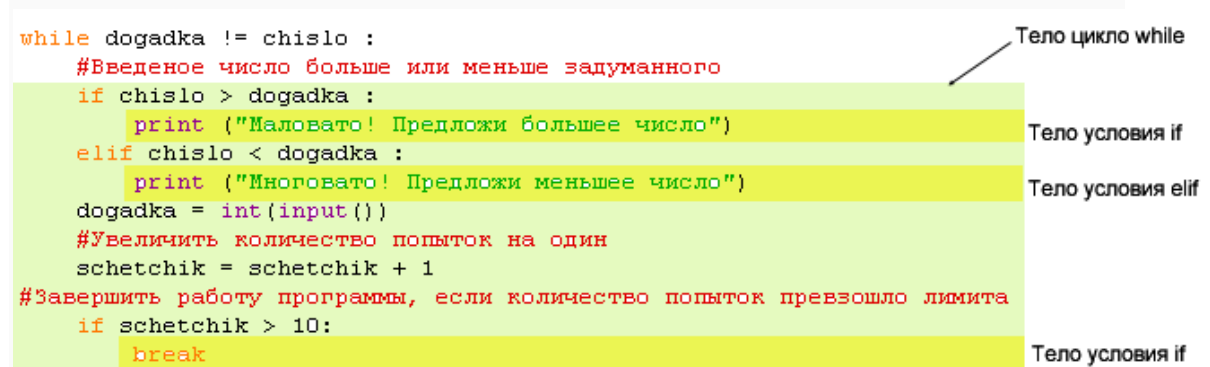
```

from pygame import *
from sys import *
VLEVO = 1
VVERH = 2
VPRAVO = 3
VNIZ = 4
init ()
okno = display.set_mode ([640,480])
foto_golova = image.load ('SnailHead.png')
foto_stolknovenije = image.load ('Bang.png')
wag_zmeiki = 10
x = 50
y = 50
dvizenie = VPRAVO
stolknovenije = False
while True:
    okno.fill([255,255,255])
    if stolknovenije :
        okno.blit(foto_stolknovenije, [x,y])
    else :
        okno.blit(foto_golova, [x,y])
    display.flip()
    time.delay(10)
    if not stolknovenije :
        if dvizenie == VVERH:
            y = y - wag_zmeiki
        elif dvizenie == VNIZ:
            y = y + wag_zmeiki
        elif dvizenie == VLEVO:
            x = x - wag_zmeiki
        elif dvizenie == VPRAVO:
            x = x + wag_zmeiki
    if x < 0 :
        x = 0
        stolknovenije = True
    if x > 600 :
        x = 600
        stolknovenije = True
    if y < 0 :
        y = 0
        stolknovenije = True
    if y > 430 :
        y = 430
        stolknovenije = True
    for e in event.get() :
        if e.type == QUIT:
            exit()
        elif e.type == KEYDOWN:
            if e.key == K_UP:
                dvizenie = VVERH
            elif e.key == K_DOWN:
                dvizenie = VNIZ
            elif e.key == K_LEFT:
                dvizenie = VLEVO
            elif e.key == K_RIGHT:
                dvizenie = VPRAVO

```

## Двойной цикл

Для начала повторим, что такое тело цикла и конструкцию *if*. Хорошим примером служит следующая картинка:



В приведенном выше примере конструкция *if* прописана внутри цикла *while*. Однако внутри цикла можно прописать другой цикл. Получаются цикл внутри цикла. Зачем это нужно? Приведу пример, а потом найдем аналогии в мире компьютерных игр.

В разделе о циклах был рассмотрен пример таблицы умножения.

```

for i in range(5):
    print(i, "x 7 = ", i*7)
  
```

Этот кусок кода выводил на экран умножение только на 7 и то, только до 5. Таблица умножения, которую учат в начальной школе, намного больше. Для улучшения нашего результата задействуем второй цикл внутри первого. Пока первый цикл увеличивает постепенно значения множимого, второй цикл проходит все заданные значения для множителя.



Открой текстовый редактор *Idle*, запиши код, сохрани и запусти. Проанализируй результат.

```

for mnozimoe in range (2,5):
    for i in range (1,11):
        print (mnozimoe, "x", i, "=", mnozimoe*i)
    print()
  
```

### Как работает двойной цикл?

- Прежде всего начинает работать внешний цикл; его первое значение 2.
- С этим значением (2) совершаются все остальные действия, прописанные во внутреннем цикле, включая вывод пустой строки на экран.

- Пустая строка выводится на экран после того, как внутренний цикл полностью завершит свою работу, т.е. переберет все значения от 1 до 10 (ты помнишь, что последнее значение, которое принимает цикл на одно меньше, чем то, которое указано в скобках на месте второго аргумента). На каждом круге прохождения цикла, внутренний цикл выпечатывает строчку из таблицы умножения. Множимое у нас фиксировано внешним циклом, а множитель увеличивается на один в результате работы внутреннего цикла.
- Когда внутренний цикл завершает работу, внешний цикл успевает дойти до приказа **print()** - вывести на экран пустую строку.
- После того, как внешний цикл завершил работу со значением 2, он продолжает работать дальше; на новом круге значение внешнего цикла равно 3.
- Внутренний цикл проходит полный круг работы.
- Внешний цикл продолжает повторять свои действия до тех пор, пока не дойдет до значения 4. Внутренний цикл совершить свои действия; на экран выводится пустая строка и на этом работа с циклами завершается.

## Примеры

Рассмотрим игру в звездочки, которая по структуре и логике схожа с таблицей умножения.



```
print ("Сколько звезд в одном ряду хочешь нарисовать?")
kol_zvezd = int(input())
print ("Сколько таких рядов хочешь нарисовать?")
rjad_zvezd = int(input())

for rad in range (rjad_zvezd):
    for zvezdi in range (kol_zvezd):
        print ("*", end=" ")
    print()
```

Проверь, что получится, если:

- изменить количество рядов и звезд;
- изменить расположение приказа *print()*;
- удали или добавь отступы;
- изменить расположение цикла *for*.

## Каким образом двойные циклы полезны при создании игр?

Двойные циклы довольно часто используют в разработке игр. Вот несколько примеров:

- Клеточки - т.н. шахматная доска - заключается в раскрашивании клеток. Создается цикл, в котором прописан приказ по раскрашиванию, и запускается 64 раза.
- Поиск оптимального решения при помощи комбинаций и пермутаций. Об этих понятиях поговорим чуть позже.

## Многократные циклы

Сделаем нашу задачу еще более интересной, добавив циклы внутри циклов. Что это значит? При помощи **одного** цикла можно нарисовать строчку звездочек, при помощи **двух** циклов - строчки звездочек. Зачем нужен третий встроенный цикл? Проанализируй, перепиши, сохрани и запусти следующий код.



```
print ("Сколько звезд в одном ряду хочешь нарисовать?")
kol_zvezd = int(input())
print ("Сколько таких рядов хочешь нарисовать?")
rjad_zvezd = int(input())
print ("Сколько блоков хочешь нарисовать?")
blok_zvezd = int(input())

for blok in range (blok_zvezd):
    for rad in range (rjad_zvezd):
        for zvezdi in range (kol_zvezd):
            print ("*", end=" ")
        print()
    print()
```



## Комбинации и пермутации

Как ранее уже было сказано, многократные циклы полезны для работы с комбинациями и пермутациями. Что эти термины означают?

- **Пермутация** или **перестановка** - изменение **последовательности** расположения определенного количества элементов.
- **Комбинация** - практически тоже самое что и пермутация, однако последовательность элементов не играет существенной роли; важны значения самих элементов.

### Рассмотрим примеры:

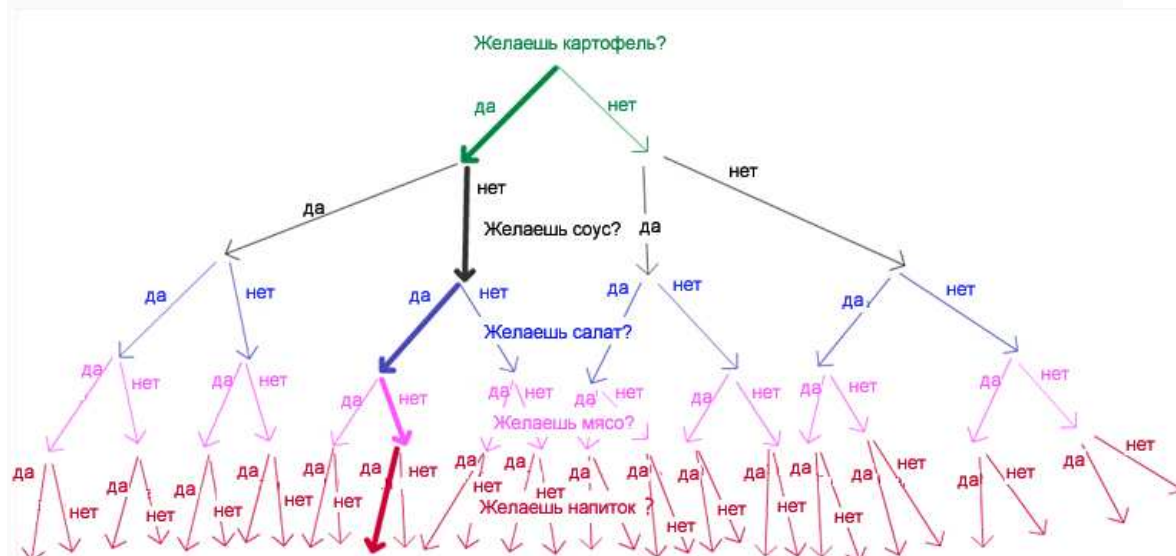
Если я попрошу тебя выбрать три числа из промежутка 1-10, то варианты могут быть следующие:

- 8, 3, 5
- 7, 2, 10

Эти два варианта называют комбинациями. Пермутация имеет место, если из трех элементов составить следующие варианты:

- 7, 2, 10
- 10, 7, 2
- 2, 7, 10

Приведем еще один конкретный пример. Предположим, в школьной столовой ты можешь выбрать себе обед из следующего меню: картофель, соус, салат, мясо, напиток. Как узнать все **комбинации**, которые ученики могут себе выбрать? Возможное решение - использовать т.н. дерево решений (см. рисунок).



На этом рисунке одна дорожка от вершины к концу составляет одну комбинацию. Жирными линиями обозначен выбор одного произвольного ученика: он взял картошку, салат и сок. Мясо и соус оставил до лучших времен.



Это же самое задание можно решить при помощи программирования:

```

print("\tkartofel \tsous \tsalat \tmyaso \tsok")
schetchik = 1
for kartofel in [0,1]:
    for sous in [0,1]:
        for salat in [0,1]:
            for myaso in [0,1]:
                for sok in [0,1]:
                    print(schetchik, "\t", kartofel, "\t\t", sous, "\t", salat,
                        "\t", myaso, "\t", sok)
                    schetchik +=1

```

"\t" - знак табуляции, который помогает красиво вывести на экран результат программы.

0 и 1 в каждом цикле означают "да" и "нет". При желании числа можно заменить словами; в этом случае не забудь про кавычки для строчковых литерал.

Как показывает результат программы, всего можно составить 32 комбинации. Другое дело, насколько они съедобные?! Но как говорят, на вкус и цвет товарищей нет.

## Пример

Дополним предыдущую программу для столовой счетчиком калорий. Программа должна показать выбранную комбинацию и количество калорий этого блюда.

Как решить эту задачу? Каждому компоненту блюда надо добавить переменную с калориями и включить ее в самый последний (внутренний) цикл.

**NB!** В примере приведенные данные взяты из реального мира!

## Список

В прошлой книге мы затронули тему типов данных - подробно рассмотрели числовые типы *integer* и *float* и строчный литерал *string*. Когда была необходимость сохранить данные в память, мы создавали переменную, которой присваивали значение. Однако реальная жизнь намного сложнее и часто за одной переменной должно стоять несколько значений, например:

- неделя состоит из 7 дней;
- контакты друга;
- черты характера учителей.



Крайне неудобно хранить каждый элемент в отдельной переменной. Для этой цели в программировании есть специальный тип данных - **список**.

### Что такое список?

**Список** - это группа или коллекция элементов, объединенных общим названием. Элементы внутри списка могут быть разных типов данных - текст, число, другие списки. Однако обычно стараются хранить в списках данные одного типа.

### Как создаются списки?

Как в случае и с другими типами данных, сначала создается переменная, которой присваиваются значения; значения присваиваются в квадратных скобках.



```
>>> cveta = ["zelenij", "krasnij", "sinij", "zeltij"]
>>> razmer_obuvi = [36,37,38,39,40,41,42]
>>> goroda = ["Tartu", "Tallinn", "Narva", "Pärnu"]
>>> ochenki = [1,2,3,4,5]
>>> balli = [45.6, 67.8, 34.9]
```

Таким образом создается т.н. готовый список. Однако можно сначала создать абсолютно пустой список и позже начать добавлять в него элементы. Элементы можно добавлять не только в пустой список, но и в список, который уже содержит данные. Следует помнить, что элементы всегда добавляются в конец списка.

Следующий пример иллюстрирует создание пустого списка и добавление элементов при помощи приказа `.append()`. Открой командную строку и попробуй создать список, который содержит числа, текстовые литералы и другой список. Пример, приведенный внизу, послужит тебе подсказкой.



```
>>> druzja = []
>>> druzja.append("Lena")
>>> print(druzja)
['Lena']
>>> druzja.append("Vadim")
>>> print(druzja)
['Lena', 'Vadim']
```



Помни, прежде чем начать добавлять элементы в список, нужно создать переменную. Без нее ничего не получится. Это как приготовление пирога. Прежде чем начать мешать компоненты, нужна сперва найти подходящего размера емкость, куда будут сыпаться компоненты.





```
>>> goroda = ["Tartu", "Tallinn", "Narva", "Pärnu"]
>>> print(goroda[0:2])
['Tartu', 'Tallinn']
>>> print(goroda[1:3])
['Tallinn', 'Narva']
>>> print(goroda[2:4])
['Narva', 'Pärnu']
>>> print(goroda[-1])
Pärnu
```

Обрати внимание, допустимы и отрицательные значения. Так быстрее всего можно получить последний элемент списка. Как получить предпоследний элемент?

## Изменить элемент списка

По порядковому номеру элемента в списке можно изменить значение элемента. Это делается так:



```
>>> goroda = ["Tartu", "Tallinn", "Narva", "Pärnu"]
>>> goroda[0] = "Valga"
>>> print(goroda)
['Valga', 'Tallinn', 'Narva', 'Pärnu']
>>>
```

## Добавление элемента в список

Кроме приказа **.append()** в Питоне есть еще и другие возможности для добавление элементов в список.

Приказ **.extend()** добавляет все элементы второго списка к первому списку, другими словами, объединяет два списка.



Приказ **.insert()** добавляет элемент на указанный порядковый номер.

```
>>> goroda = ["Tartu", "Tallinn", "Narva", "Pärnu"]
>>> goroda[0] = "Valga"
>>> print(goroda)
['Valga', 'Tallinn', 'Narva', 'Pärnu']
>>> goroda.extend(["Tartu", "Viljandi", "Rakvere"])
>>> print(goroda)
['Valga', 'Tallinn', 'Narva', 'Pärnu', 'Tartu', 'Viljandi', 'Rakvere']
>>> goroda.insert(2, "Kuressaare")
>>> print(goroda)
['Valga', 'Tallinn', 'Kuressaare', 'Narva', 'Pärnu', 'Tartu', 'Viljandi', 'Rakvere']
>>> |
```

Не забудь, что исчисление элементов в списке начинается с нуля. Это значит, что добавляя элемент на 2 место, элемент получает порядковый номер 1.

## Удаление элемента из списка

Как удалить элемент из списка? Есть три возможности `.remove()`, `del` ja `.pop()`. Это не синонимы.

- **`.remove()`** удаляет элемент по его имени
- **`del`** удаляет элемент по его порядковому номеру
- **`.pop()`** удаляет последний элемент списка; этот приказ можно применить для удаления элемента из списка по его порядковому номеру.

```
>>> print(goroda)
['Valga', 'Tallinn', 'Narva', 'Pärnu', 'Tartu', 'Viljandi', 'Rakvere']
>>> goroda.insert(2, "Kuressaare")
>>> print(goroda)
['Valga', 'Tallinn', 'Kuressaare', 'Narva', 'Pärnu', 'Tartu', 'Viljandi', 'Rakvere']
>>> goroda = ([ 'Valga', 'Tallinn', 'Kuressaare', 'Narva', 'Pärnu', 'Tartu', 'Viljandi', 'Rakvere'])
>>> goroda.remove("Valga")
>>> print(goroda)
['Tallinn', 'Kuressaare', 'Narva', 'Pärnu', 'Tartu', 'Viljandi', 'Rakvere']
>>> del goroda[2]
>>> print(goroda)
['Tallinn', 'Kuressaare', 'Pärnu', 'Tartu', 'Viljandi', 'Rakvere']
>>> goroda.pop()
'Rakvere'
>>> print(goroda)
['Tallinn', 'Kuressaare', 'Pärnu', 'Tartu', 'Viljandi']
>>> goroda.pop(2)
'Pärnu'
>>> print(goroda)
['Tallinn', 'Kuressaare', 'Tartu', 'Viljandi']
```

## Поиск элемента

Хорошо, когда данные можно добавлять, хранить, удалять. А если нужно найти значение, что тогда делать? Все зависит от цели поиска:

- найти, есть ли такой элемент в списке или нет;
- найти порядковый номер элемента, которое хранит нужное значение.

### Ключевое слово *in*

Разберем сначала первый пункт: есть ли элемент в списке.



```
>>> goroda
['Tallinn', 'Kuressaare', 'Tartu', 'Viljandi']
>>> if "Tallinn" in goroda:
    print("da")
else: print("net")

da
>>> "Tartu" in goroda
True
>>> "Paldiski" in goroda
False
>>> |
```

Как видно из примера, ключевое слово *in* возвращает значение *True* или *False*. Очень удобный результат для того, чтобы использовать его в конструкциях *if*.

## Поиск нахождения элемента или поиск индекса

Поиск порядкового номера элемента в списке называют поиском **индекса** элемента. Для этого существует приказ **.index()**:



```
>>> goroda
['Tallinn', 'Kuressaare', 'Tartu', 'Viljandi', 'Narva']
>>> goroda.index("Narva")
4
>>> if "Viljandi" in goroda:
    print(goroda.index("Viljandi"))

3
```

## Сортировка списка

Элементы в списках находятся в такой последовательности, как их добавили в список. У каждого элемента свой уникальный индекс. Индекс меняется только при использовании приказов **.insert()**, **.append()**, **.remove()** или **.pop()**.

Расположение элементов в таком порядке, как их поместили в список, не всегда бывает удобным. Задачу можно решить при помощи сортировки. Например, отсортировать список по возрастанию или убыванию значений или отсортировать элементы списка в алфавитном порядке значений.

Для сортировки списков используется приказ **.sort()**

```
>>> goroda=[['Valga', 'Tallinn', 'Kuressaare', 'Narva', 'Pärnu', 'Tartu', 'Viljandi', 'Rakvere']]
>>> goroda.sort()
>>> goroda
['Kuressaare', 'Narva', 'Pärnu', 'Rakvere', 'Tallinn', 'Tartu', 'Valga', 'Viljandi']
>>> balli = ([45.6, 67.8, 34.9])
>>> balli.sort()
>>> balli
[34.9, 45.6, 67.8]
>>>
```

Списки можно отсортировать и в обратном порядке, т.е. в убывающем порядке, при помощи команды **.reverse()**

```
>>> goroda
['Kuressaare', 'Narva', 'Pärnu', 'Rakvere', 'Tallinn', 'Tartu', 'Valga', 'Viljandi']
>>> goroda.reverse()
>>> goroda
['Viljandi', 'Valga', 'Tartu', 'Tallinn', 'Rakvere', 'Pärnu', 'Narva', 'Kuressaare']
>>> balli
[34.9, 45.6, 67.8]
>>> balli.reverse()
>>> balli
[67.8, 45.6, 34.9]
>>> |
```

## Что ты узнал?

На этой неделе ты пополнил свой багаж знаний еще несколькими важными аспектами программирования:

- двойные и вложенные циклы
- зачем нужны двойные циклы?
- почему двойные циклы незаменимы?
- комбинации и пермутации
- как комбинации связаны со вложенными циклами?
- что такое дерево решений?
- что такое список?
- как создать список?
- как добавляются и удаляются элементы из списка?
- как найти нужный элемент из списка?
- как узнать индекс (порядковый номер) нужного элемента?
- как быстро получить значение последнего элемента списка?
- сортировка списка



Если ты уяснил материал, рассмотренный в течение трех недель, разобрал все примеры и выполнил домашние задания, то большую теоретическую часть работы ты уже выполнил и у тебя достаточно знаний для создания простеньких игрушек. Вперед к следующей книге и новым приключениям!

## НЕДЕЛЯ 4: ФУНКЦИИ И ОБЪЕКТЫ

### И снова игра Змейка



И снова здравствуй, новых открытий тебе на этой неделе! На этот раз опять для тебя приготовлены три новых дополненных версии игры Змейка. Первую из них - **Версию4** - сохрани к себе на компьютер под именем *Vol4.py*. Попробуй самостоятельно разобраться в коде, запусти его. На следующей неделе рассмотрим различные блоки с детальными разъяснениями. Для этой части тебе понадобится еще звуковой файл. Его можно найти по адресу <http://math.ut.ee/~kull/kokkuporge.wav> и возьми также иконки головы и



столкновения:

**NB!** Код можно посмотреть в отдельном [окне](#).

### Версия 5



Теперь игра начинает приобретать новые очертания. Изучи различия в версиях 4 и 5, внеси изменения к себе в файл. Новый файл сохрани с названием *Vol5.py*. Тебе понадобятся звуковой файл, который найдёшь по адресу <http://math.ut.ee/~kull/soodud.wav>, а также картинка мышки и иллюстрация для съеденной мышки:



**NB!** Код можно посмотреть в отдельном [окне](#).

### Версия 6



Сюрприз! Что-то изменено в программном коде. Но что именно?

**NB!** Код можно открыть в отдельном [окне](#).

## Функции

При написании больших программ, как например игры, очень быстро код становится длинным и сложным. Трудно становится запомнить используемые конструкции, комбинации и уровень читаемости кода снижается. Ты это уже успел почувствовать при переписывании кода игры Змейка - код с каждой версией становился всё длиннее и запутаннее.


Лучше всего код разбить на небольшие части. В таком случае у программиста есть обзор, он знает что делает тот или иной кусочек кода, каков будет результат выполнения задания данной частью. К тому же написание одного кусочка намного проще, чем писать всё программу за один раз. На самом деле ты уже занимался разделением кода на части при переходе игры Змейка от Версии3 к Версии4. Версия3 уже становилась довольно длинной, но Версия4 была намного лучше структурирована, компактнее и понятнее.

### Что же изменилось в Версии4?

Мы воспользовались термином функция. **Функция** есть не что иное как один кусочек кода из всей большой программы, которому присвоено своё имя и которое может возвращать (выдавать) результат своей работы. Функции часто называют также **подпрограммами**, так как они ведут себя как маленькие самостоятельные программки. Функция - это сборник команд, который что-то выполняет. Функции можно рассматривать и как строительный материал при постройке дома. У каждого материала есть своё предназначение, он обладает своими специфическими свойствами, своей структурой и т.д. Из многих различных и одинаковых кусочков материала можно при правильном подходе построить любое строение. Также и с функциями - при искусной комбинации можно собрать в единое целое какую угодно программу.

### Создание функции

Функция создаётся или **дефинируется** с помощью ключевого слова **def**. Ты наверняка обратил на это внимание в Версииб игры Змейка. За ключевым словом **def** записывается **имя функции** (при выборе имени для функции стоит исходить из тех же правил, что и при работе с переменными) и в самом конце **скобки**. Внутри скобок можешь, но не должен, что-то записывать. Это зависит от особенностей работы данной функции, но об этом чуть попозже. А сейчас открой редактор *Idle* и потренируйся в создании функции.

	<pre>def panda ():     print ("Мне нравятся панды!")     print ("панды "*2)     print ("фуфф "*5)     print ("Хватит! Мартышки лучше") panda ()</pre>	<p>Определим функцию <i>panda</i>. Данная функция выводит только четыре строчки текста, больше ничего. Для запуска функции необходимо вызвать</p>
---	---	---



		её по имени, что мы и делаем в последней строке      данного примера.      Особое внимание      обрати именно на <b>скобки</b> .
--	--	--

Но для чего ещё необходимо отделять часть команд от большого кода и давать им какое-то общее имя? Неужели действительно только ради улучшения читабельности?

## Вызов функции

Отвечу сразу на заданные в конце прошлой главы вопросы: нет, конечно же функции существуют не только для того, чтобы дать какой-то заголовок для различных частей кода, как это делается с главами книг. У функций более значимое применение.

Благодаря функциям можно записать какой-то кусочек кода только один раз и, если необходимо этой частью воспользоваться в различных местах кода, то можно в программе просто обратиться (вызвать) к соответствующей функции по имени. Функцию можно вызвать в любом месте программы и столько раз, сколько потребуется. При этом не надо писать повторяющийся кусочек кода снова и снова.

Рассмотрим один простой пример:



```
import random

chislo1 = random.randint(1,10)
chislo2 = random.randint(1,10)

def molodec():
    print ("Правильный ответ")
    print ("Молодец!")

def zalko():
    print ("К сожалению, ты дал неправильный ответ :(")

print ("Сколько будет", chislo1, "+", chislo2)
otvet1 = int(input())

if otvet1 == chislo1 + chislo2:
    molodec()
else: zalko()

print ("Сколько будет", chislo1, "-", chislo2)
otvet1 = int(input())

if otvet1 == chislo1 - chislo2:
    molodec()
else: zalko()
```

В данном примере дефинируется две функции - *molodec* и *zalko*. Они вызываются в двух *if*-предложениях, первый раз при проверке сложения двух чисел, второй раз при проверке разницы двух чисел. Хотя данные функции очень маленькие и по сути дела не делают ничего особенного, однако можно почувствовать некую экономию времени.

Конечно, можно было бы эти два предложения просто скопировать в оба *if*-предложения, но это сделает наш код менее читабельным. Зачастую содержание функций намного сложнее, чем использование команды *print*.


**NB!** Есть ещё одна причина использования функций. Бывает необходимым использовать несколько раз один и тот же кусочек кода в различных местах программы. Предположим, что без использования функции мы прописали бы эти строки кода несколько раз (например, 10 - 100). Через какое-то время обнаруживается, что строки содержали типичную ошибку. Теперь для исправления необходимо просмотреть всю программу и исправить все места, где была допущена ошибка (в 10 - 100 случаях). Конечно, это возьмёт время и будет очень утомительно. В случае использования функции было бы достаточно просто исправить ошибку внутри такого кусочка кода. Эти бы изменения отразилось бы сразу во всей программе.

## Аргументы функции

**Аргументы функции** - это информация, которую можем передать функции, записав её в скобках. Такая передача информации для функции называется **передачей функции аргументов**.

Для того, чтобы запутать тебя ещё больше, отметим, что некоторые программисты называют добавленную в скобках информацию **параметром**. На самом деле, большой разницы между этими двумя терминами нет. Различия обусловлены особенностью англоязычных слов. Если я передаю информацию функции во время её вызова, то такую информацию называют аргументом, но, если я использую полученную информацию внутри функции для выполнения чего-то, то это называется параметром.

Рассмотрим тот же пример с вычислениями, но с использованием аргументов функций.



```
import random

chislo1 = random.randint (1,10)
chislo2 = random.randint (1,10)

def molodec(otvet):
    print (otvet, " - это правильный ответ")
    print ("Молодец!")

def zalgo(vastus):
    print (otvet, ", к сожалению, ты дал неправильный отве

print ("Сколько будет", chislo1, "+", chislo2)
otvet = int(input())

if otvet == chislo1 + chislo2:
    molodec(otvet)
else: zalgo(otvet)

print ("Сколько будет", chislo1, "-", chislo2)
otvet = int(input())

if otvet == chislo1 - chislo2:
    molodec(otvet)
else: zalgo(otvet)
```

В данном примере видно, что функция использует в своей работе переданную ей информацию (аргумент). Аргумент *otvet* используется внутри функции в команде *print*. Аргумент передаётся функции в момент её вызова.

В этом примере при определении функции и её вызове используется одно и тоже название переменной *otvet*. Хотя этого не рекомендуется делать. Имена переменных (как и имена аргументов) внутри функций могут отличаться от тех, которые используются при вызове функции.

У функции может быть несколько аргументов, для каждого из них необходимо присвоить имя. При вызове функции важно следить за последовательностью (порядком) значений, записываемых внутри скобок.



Посмотри видео.

## Возвращение результатов работы функции

До сих пор в виде примеров использовались такие функции, которые сразу что-то выводят с помощью команды *print()*. Но это не является обычным использованием функций. Часто функции используются для вычисления какого-либо значения, для изменения объекта, для добавления чего-то. Всё это особенно не касается команды вывода результата на экран. Если функция вносит какое-то изменение или выполняет расчёт, то как же пользователь узнает о результатах?

### Возвращение результатов работы функции

Для этого внутри функции используется ключевое слово **return**. Сразу за *return*-словом необходимо записать результат, который необходимо вернуть. Обязательно изучи следующий пример:

```
import math

print ("Эта программа вычисляет длину окружности")
print ("Каков радиус окружности (см) ?")
radius = float (input())

def dlina_okruznosti (r):
    dlina = round(2*math.pi*r,2)
    return dlina

print ("Длина окружности будет", dlina_okruznosti (radius))
```

В данном примере в последней команде *print()* вызывается функция вычисления длины окружности. Обрати внимание, что аргументом функции является то значение радиуса, которое спросили у пользователя. Однако внутри функции имя аргумента, т.е. параметр, задан через *r*. Задачей функции является вычисление длины окружности и возвращение результата с помощью команды *return*. Поэтому с помощью команды *print* не печатается имя или содержание функции, а только **вычисленная функцией длина окружности**.

## Локальные и глобальные переменные

```
import math

print ("Эта программа вычисляет длину окружности")
print ("Каков радиус окружности (см)?")
radius = float (input())

def dlina_okruznosti (r):
    dlina = round(2*math.pi*r,2)
    return dlina

print ("Длина окружности будет", dlina_okruznosti (radius))
```

В данном примере стоит особое внимание обратить на различные имена переменных внутри и за пределами функции, хотя эти переменные в обоих случаях обозначают радиус окружности. Это связано с понятиями **локальная** и **глобальная переменная**.

**Локальной переменной** называют такую переменную, которая располагается внутри функции. В данном примере переменные **r** и **dlina** будут локальными. Переменная же **radius** будет **глобальной**. В чём между ними различия, почему одну называют локальной, а другую глобальной переменной?

Разница между **локальной** и **глобальной** переменными заключается в **области их воздействия**. В *Python*'е все переменные, расположенные внутри функции, будут локальными и за пределами функции их нельзя будет использовать. Если бы пришлось дописать в конец программы ещё одну команду *print* и в скобках записать функциональную переменную **dlina**, то появилось бы сообщение об ошибке. В ней бы говорилось, что программа не узнаёт такую переменную. В *Python*'е продумано так, что до запуска функции на самом деле не существует внутри функции какой-либо переменной. Только после того как функция будет вызвана, *Python* создаёт соответствующие (локальные) переменные. В данном случае это **r** и **dlina**, которым и присваивается какое-то значение. **r** получит значение от аргумента радиус. **dlina** находится в результате произведений переменных **pi** и **r**. Как только функция вернёт результат вычисления, *Python* удаляет все использованные в функции переменные - память освобождается и таким образом для программы переменные функции становятся опять несуществующими.

**Глобальные** же переменные существуют на протяжении всей работы программы и их можно использовать везде, в том числе внутри функции.

**NB!** Глобальные переменные можно изменять в любом месте программного кода и в любое время. Однако внутри функции этого сделать не удастся, можно использовать только её значение. Почему же так? Например, как только понадобится изменить значение глобальной переменной **radius** на новое (например, *radius* = 20), то на самом деле такая строчка глобальную переменную не изменит. Дело в том, что все переменные, находящиеся внутри функции, создаются заново даже тогда, когда её имя совпадает с именем глобальной переменной. Таким образом, просто создаётся локальная переменная с таким же именем, что и у глобальной переменной.

### Глобальная переменная внутри функции

Если же понадобится внутри какой-то функции изменить глобальную переменную, то перед именем глобальной переменной необходимо записать **global**. В таком случае, функция действительно изменит глобальную переменную и не создаст временную одноимённую локальную переменную.

Длинный текст, но посмотрим на примере работу с переменными.

Посмотри видео.



## Объекты

Уже некоторое время рассматриваем, как можно организовать свои данные и различные части кодов в своей программе. Одинаковые переменные научились соединять с помощью списков, кусочки кода с помощью функций.

Теперь же поговорим о следующем шаге - **объектах**. С помощью них под одним именем можно расположить вместе как функции, так и переменные. Объекты довольно распространены в программировании, и их используют во многих языках. Если один раз ты понял идею работы с объектом, то сможешь разобраться с любым другим объектно-ориентированным языком.

### Для чего говорить об объектах?

Как только разберёмся с этой частью, ты поймёшь, почему же *Python* называют объектно-ориентированным языком. Здесь все вещи принадлежат объектам, хотя сразу об этом можно и не догадаться. До сих ни в одном из наших заданий не было явно заметно работы с объектом, хотя уже на следующей неделе начнём заниматься графикой.



Тогда без понимания термина объекта практически не обойтись. Данная глава будет своего рода введением в материалы следующей недели.

### Что такое объект?

Возьмём в качестве примера книгу, которую можно рассматривать как объект. Как это понимать?

- Каждый раз с объектом книга мы можем что-то делать - например, читать, дарить, положить на полку, закрыть, открыть и т.д.
- Мы можем описать свойства данного объекта - количество страниц, твёрдая или мягкая обложка, размеры, форма, издательство, заголовок, жанр и др.

В программировании с объектами выполняются те же вещи - **действия**, которые можно совершить над объектом, и **описать** его. В *Python*'е данные, которые ты знаешь об объекте, называют **атрибутами**, а действия, которые ты можешь делать с объектами - **методами**.

### Объект = атрибуты + методы

Например, если бы в *Python*'е необходимо было бы создать объект книга, то у неё могли бы быть:

- следующие **атрибуты**: `kniga.koli4estvo_stranic`, `kniga.razmer`, `kniga.forma` и т.д.
- и такие **методы**: `kniga.chtenije()`, `kniga.otkritije()`, `kniga.zakritije()` и др.

Обратил ли ты внимание на использование точки? Вспомни, мы неосознанно использовали в своих кодах с примерами довольно много объектов. Например, при составлении программы по измерению времени использовали **`time.sleep()`**, а при добавлении данных в лист задействовали **`ptichki.append()`**. Таким образом, что *time*, что *ptichki* были для *Python*'а объектами, а после точки располагалась метод для объекта.

В роли **атрибутов** могут выступать всевозможные переменные и их списки (числа, описания и т.д.), а в роли **методов** обычно выступают функции, которые принадлежат данному конкретному объекту. Атрибуты - это информация об объектах, а методы - это действия над объектами.

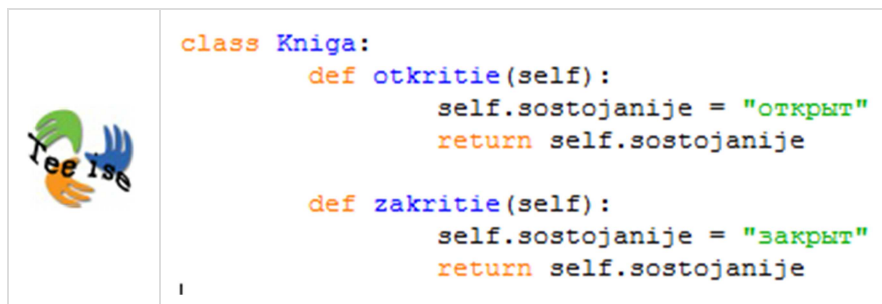
Атрибуты же на самом деле не что иное, как обычные переменные. Им также присваиваются значения. В программном коде их используют также как обычные переменные, о которых уже успели поговорить ранее. Единственное отличие заключается в том, что атрибуты используют всегда вместе с именем объекта и для объединения служит **точка**.

## Создание объекта

Создание объекта проходит в два этапа:

- для начала необходимо определиться (дефинировать) атрибуты, принадлежащие объекту, и методы. Чем-то этот пункт напоминает составления плана работы по строительству дома. Дома как такового ещё нет, но чертёж уже существует. Такой план работы на языке *Python*'а называют **классом**.
- во время второго шага необходимо взять класс (т.е. план) и создать по нему настоящий объект (т.е. построить реальный дом). На основании класса можно создать всевозможные объекты - как и по одному и тому же плану можно создать несколько одинаковых домов. Из классов созданные объекты в программировании называют **экземплярами**.

Изучим работу одного класса и создание экземпляра. Попробуй и у себя в *Idle* поработать с одним классом.



В выше приведённом примере был создан простой класс *Kniga*, где используется два метода - открытие и закрытие. Они должны бы были изменять вид объекта на экране - книга либо открыта, либо закрыта.

## А где же атрибуты?

Атрибуты не принадлежат целому классу, а только непосредственно какому-то экземпляру. То есть по типовому плану построенный дом может иметь свой цвет и при строительстве могут быть использованы другие отделочные материалы. У каждого объекта может быть только



ему присущее описание, но экземпляры одного и того же класса всегда ведут себя одинаково.

## Создание экземпляра

Как было описано ранее, создание класса - это создание чертежа. Теперь же надо заняться строительством. Для создания экземпляра его необходимо в программе записать и делается это следующим образом:

```
class Kniga:
    def otkritie(self):
        self.sostojaniye = "открыт"
        return self.sostojaniye

    def zakritie(self):
        self.sostojaniye = "закрыт"
        return self.sostojaniye

rasskaz = Kniga()
rasskaz.sostojaniye = "закрыт"
rasskaz.razmer = (10, 15)

print ("Вышел небольшой рассказ.")
print ("Размер рассказа", rasskaz.razmer[0], "x", rasskaz.razmer[1], "см.")
print ("Мой рассказ сейчас", rasskaz.sostojaniye)
print ("Открываю рассказ.")
print ("Теперь он", rasskaz.otkritie())
print ("Закрою книгу.")
print ("Рассказ теперь", rasskaz.zakritie())
```



Посмотри видео.

## Установка исходного состояния объекта

В описание класса атрибуты не вписываются, но очень сложно и долго было бы каждый раз при создании объекта присваивать ему аргументы. Эту проблему можно решить с помощью специального метода `__init__()`, внутри которого можно для каждого создаваемого объекта записать так называемые исходные значения, параметры. `__init__()` - это специальный метод (перед и после имеется два нижних подчёркивания), который запускается каждый раз, как создаётся новый экземпляр. Таким образом, объект получает первые значения, первое описание. *init* происходит от английского слова *initializing*, что и обозначает инициализация, обозначение исходного состояния.

Изменим пример с книгой таким образом, чтобы не пришлось описывать объект отдельно:



```
class Книга:
    def __init__(self, sostojanije, razmer):
        self.sostojanije = sostojanije
        self.razmer = razmer

    def otkritie(self):
        self.sostojanije = "открыт"
        return self.sostojanije

    def zakritie(self):
        self.sostojanije = "закрыт"
        return self.sostojanije

rasskaz = Книга("закрыт", (15,10))

print ("Вышел небольшой рассказ.")
print ("Размер рассказа", rasskaz.razmer[0], "x", rasskaz.razmer[1], "см.")
print ("Мой рассказ сейчас", rasskaz.sostojanije)
print ("Открываю рассказ.")
print ("Теперь он", rasskaz.otkritie())
print ("Закрою книгу.")
print ("Рассказ теперь", rasskaz.zakritie())
```

В результате внесённых изменений работа абсолютно не изменилась, но мы избавились от необходимости для каждого объекта задавать исходные состояния, т.к. прописали их в виде параметров в скобках при создании объекта.



Посмотри видео.

## \_\_str\_\_()

В классах *Python*'а имеется ещё один скрытый метод, который также как и `__init__()` всегда готов к запуску. Этим методом является `__str__()`. На примере книги изучи, что этот метод делает. Чтобы это узнать, необходимо в конец программного кода дописать ещё одну строку `print(rasskaz)`. В результате получим что-то такого вида:

```
Вышел небольшой рассказ.  
Размер рассказа 15 x 10 см.  
Мой рассказ сейчас закрыт  
Открываю рассказ.  
Теперь он открыт  
Закрою книгу.  
Рассказ теперь закрыт  
<__main__.Kniga object at 0x024BCF90>
```

Последняя строка в этом примере и есть результат запущенного метода `__str__()`. Он выдаёт информацию о спрашиваемом объекте (в данном случае, о рассказе). Эта информация была спрошена с помощью *print* команды и по умолчанию (так как мы не уточнили, какая информация нам нужна) получили именно такую последнюю строчку, которую мы видим на вышеприведённом примере. Что она означает?

- первое: место, куда определяется данные экземпляр - в конкретном случае это `__main__`, что означает основную программу;
- второе: имя класса - *Kniga* - рассказ является объектом книги;
- третье: набор цифр и букв означает адрес в памяти, где в компьютере пользователя располагается данный объект.

В принципе, довольно бессмысленная информация для обычного пользователя программы. Поэтому если необходимо получить какую-то другую информацию о данном объекте, например, описание свойств объекта, тогда метод `__str__()` необходимо переопределить:

```

class Kniga:
    def __init__(self, sostojanije, razmer):
        self.sostojanije = sostojanije
        self.razmer = razmer

    def otkritie(self):
        self.sostojanije = "открыт"
        return self.sostojanije

    def zakritie(self):
        self.sostojanije = "закрыт"
        return self.sostojanije

    def __str__(self):
        soobwenije = ""
        и в данный момент я "" + self.sostojanije + ".\n" \
        "Мои размеры " + str(self.razmer[0]) \
        + "x" + str(self.razmer[1]) + "см."
        return soobwenije

rasskaz = Kniga("закрыт", (15,10))

print ("Вышел небольшой рассказ.")
print ("Размер рассказа ", rasskaz.razmer[0], "x", rasskaz.razmer[1], "см.")
print ("Мой рассказ сейчас", rasskaz.sostojanije)
print ("Открываю рассказ.")
print ("Теперь он", rasskaz.otkritie())
print ("Закрою книгу.")
print ("Рассказ теперь", rasskaz.zakritie())

```



Посмотри видео.

## self

Везде в методах класса используется слово **self**, но что оно делает, что это такое?

*Self* - это очень хитрое слово. Дело в том, что дефинирование класса означает только создание плана, чертежа, по которому потом будут создавать объекты. Объектов может быть сколько угодно одинаковых, только с разными именами. Отсюда и прослеживается необходимость в самом себе.

## Как?

Если у нас несколько объектов книг, например, рассказ, комикс, роман и т.д., то для каждого из этих объектов действуют одни и те же методы, которые описаны в классе. Откуда же тогда программа при запуске

метода должна знать, какие параметры экземпляра она должна изменить? По этой причине и взято в использование дополнительная переменная *self*, которая получает значение от объекта, которые её и вызвал. Если в роли вызывающего выступает *rasskaz.otkritie()*, то *self = rasskaz*. Если же вызывающим является *roman.zakritie()*, то *self = roman* и т.д.

На самом деле вместо имени переменной *self* можно использовать любое другое слово, но всё-таки лучше придерживаться традиций, так как это позволяет и тебе, и другим лучше понимать написанные строки кода.

## Чем объекты хороши?

Кроме того, что можно классифицировать различные функции и переменные, есть ещё парочка вещей, без которых жизнь программиста немислима.

### Первая из них - какой-то полиморфизм

Заумное слово, но простое объяснение. Суть: возможность использовать точно такие же имена функций в составе различных классов. Предположим, что написана программа для вычисления площади различных геометрических фигур. В такой программе для каждой различной фигуры определён отдельный класс, что является логичным - у каждой фигуры имеются свои свойства, свои формулы вычисления площади и объёма. Но в каждом классе я могу использовать один и тот же метод с именем *plowad()*, так как он принадлежит разным классам. Запутать или перепутать эти методы нельзя, так как их применяют только к объектам своего класса. Например, *treugolnik.plowad()*, *kvadrat.plowad()*, *prjamougolnik.plowad()* и т.д. Название фигуры перед точкой говорит о том, из какого класса метод *plowad()* будет вызываться. Поэтому и получается возможным вычислить для каждой фигуры именно её площадь.

### Вторая - это наследование

В объектно-ориентированном программировании можно задать так называемые **подклассы**. Что это нам даёт? Это даёт нам то, что в самом высоком классе у меня записаны только те методы, которые принадлежат всем объектам из этого класса. В подклассы можно будет записать более специфические методы, которые у всех объектов могут отсутствовать.

Например, зачастую в играх необходимо собирать различные вещи: мячики, птичек, монетки и т.д. Все они могут принадлежать одному большому классу *ObjektIgri*, у которого есть атрибут *nazvanije* (*mjachik*, *monetka*) и метод *sobrat()*. Часто некоторые вещи в игре имеют большее значение, чем другие. Например, благодаря монеткам, можно будет потом наверняка совершить какое-то действие, поэтому для них необходимо создать подкласс *Moneta*, которая будет наследовать все методы от своего "начальника", но при этом можем ему подписать метод *potratit()*, что позволит совершить с помощью денег какие-то действия.

```
class ObjektIgri:
    def __init__(self, name):
        self.name = name

    def sobrat(self, igrok):
        # сюда необходимо записать код, который
        # добавит вещь в коллекцию игрока

class Moneta(ObjektIgri):
    def __init__(self, stoimost):
        ObjektIgri.__init__(self, "монета")
        self.stoimost = stoimost

    def potratit(self, pokupatel, prodat):
        # сюда необходимо записать код, который
        # даст монету продавцу и положит полученный
        # от продавца товар в коллекцию покупателя
```

## Пример



В видео приводится пример работы с объектом.

## Что ты изучил?

В рамках этой недели ты изучил то, что понадобится для работы с графикой на следующей неделе. Можно будет приступить к созданию более интересных программных кодов. Итак, на этой неделе ты изучил:

- создание и использование функций;
- как вызывать и вернуть результат работы функции;



- разницу между локальной и глобальной переменными;
- как изменить глобальную переменную внутри функции;
- что такое объекты;
- как создаются объекты;
- что такое атрибуты и методы;
- как определить класс;
- для чего нужны специальные методы `__init__()` и `__str__()`;
- что такое явление полиморфизма;
- как создаются подклассы.



## НЕДЕЛЯ 5: ГРАФИКА И АНИМАЦИЯ

### Графика

Вот мы и добрались до раздела Графика. Используя её, можно сделать работу более интересной и красочной, но без изучения и понимания основ программирования было бы трудно разобраться в данной теме. Поэтому мы и прошли такой, возможно кажущимся тебе длинным, путь.

Для работы с графикой нам понадобится готовый пакет программы (**Pygame**), который выводит созданное на экран, позволяет чему-то двигаться, создавать рисунки, добавлять звук и т.д.

#### Для чего использовать *Pygame*?

Без *Pygame* запуск графических компонентов был бы очень сложным процессом, так как пришлось бы учитывать работу различных компонентов компьютера, а не просто работу с памятью. Понадобилось бы изучать версии операционных систем, так как для каждой системы может быть предусмотрен свой принцип работы со звуком и картинкой, уточнять параметры звуковой и графической карт, учитывать скорость работы процессора и т.д. *Pygame* же "обучен" работать с различными системами и картами, поэтому большая часть работы за нас уже сделана. Нам просто остаётся наслаждаться плодами труда программистов. К тому же данный пакет доступен бесплатно.

У тебя уже *Pygame* установлен на компьютер, и первые попытки работы с ней были уже сделаны при работе с материалом прошлой и позапрошлой недель. На этой неделе рассмотрим более детально работу с данным пакетом, как составляются компоненты, как генерируется окно, как и почему вещи передвигаются на экране с помощью клавиш или мышки.

### Создание окна

Первая вещь при создании игры - это реализация окна, области, где будет проходить сама игра. Для этого надо написать три строчки:



```
import pygame
pygame.init()
# установка размеров классического окна
okno = pygame.display.set_mode([640,480])
```



При запуске данного кода можно увидеть чёрное окно, которое в будущем будет игровым полем. Для лучшей работы игры в код программы лучше дописывать такой цикл, который бы проверял, когда пользователь закрывает окно игры. Также стоит перед `sys.exit()` записать в `while`-цикл командную строку `pygame.quit()`. Она позволяет уменьшить риск возникновения конфликта между *Idle* и *Pygame*.

Напишем цикл, который всегда является истинным:

```
import pygame, sys
pygame.init()
# установка размеров классического окна
okno = pygame.display.set_mode([640,480])

while True:
    for i in pygame.event.get():
        if i.type == pygame.QUIT:
            sys.exit ()

# Цикл длится бесконечно, т.к. True всегда будет True.
# Внутри цикла вторым циклом проверяется последовательность
# событий Pygame'a. Если тип события совпадает с ним (т.е.
# нажали кнопку выхода (QUIT), то вся система закрывается
```

**NB!** Если ты запустишь данный кусочек программы через *Idle*, то заметишь необходимость закрыть *Pygame* с применением "силы". Это обусловлено конфликтом между *Idle* и *Pygame*. В чём же он заключается? Всё дело в том, что и в *Idle*, и в *Pygame* запускается цикл контроля событий и синхронизировать их между собой очень непросто.

## Рисуем внутри окна

Рисование в *Pygame*'е используют в основном для создания фона, а не для создания героев. Герои, которые бегают в игре, или вещи, которые необходимо собирать, на самом деле являются уже готовыми картинками. Они созданы заранее с помощью какого-нибудь графического редактора. Здесь стоит обратить внимание, что нарисовать надо героев в различных позах - когда он стоит, двигается в правую или левую сторону, ползёт и т.д. В *Pygame*'е под рисованием подразумевается работа с цветом, фигурами, линиями и т.д. Рассмотрим, как это происходит.

### Первые попытки нарисовать что-нибудь

Прежде всего поменяем цвет экрана на белый и затем добавим красный круг, зелёный прямоугольник и синюю линию. Для отдельного

тестирования каждой создаваемой фигуры необходимо будет записать строку **pygame.display.flip()**, которая будет обновлять картинку экрана.

```
import pygame, sys
pygame.init()
# установка размеров классического окна
okno = pygame.display.set_mode([640,480])
# заполняем белым цветом
okno.fill([255,255,255])

# рисуем красный круг на экране,
# координаты: 50 по оси x и 50 по оси y,
# радиус будет 25, величина рамки 0
pygame.draw.circle(okno, [255,0,0], [50,50],25,0)

# рисуем зелёный прямоугольник на экране,
# координаты: 100 по оси x и 50 по оси y,
# ширина 150, высота 80, величина рамки 0
pygame.draw.rect(okno, [0,255,0], [100,50,150,80],0)

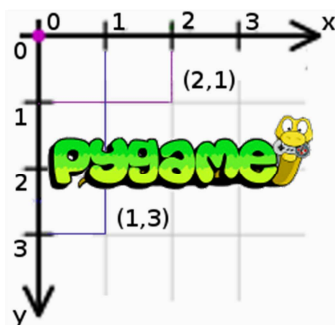
# рисуем синюю линию на экране,
# координаты: 200 по оси x и 200 по оси y,
# конечная точка линии находится в отметке
# 400 по оси x и 400 по оси y, толщина линии 3
pygame.draw.line(okno, [0,0,255], (200,200), (400,400),3)

# Pygame сначала создаст экранную картинку и с помощью
# команды flip() обновит весь экран за один раз
pygame.display.flip()

while True:
    for i in pygame.event.get():
        if i.type == pygame.QUIT:
            sys.exit ()
    # Цикл длится бесконечно, т.к. True всегда будет True.
    # Внутри цикла вторым циклом проверяется последовательность
    # событий Pygame'a. Если тип события совпадает с ним (т.е.
    # нажали кнопку выхода (QUIT), то вся система закрывается
```

Со всеми возможностями **pygame.draw** можно ознакомиться на веб-странице по адресу: <http://www.pygame.org/docs/ref/draw.html>. Там напротив каждой функции указано для чего она может использоваться, какие могут быть аргументы, что и в каком порядке необходимо записать в скобках и т.д.

## Координаты окна



Думаешь, что умеешь читать координатные значения? Посмотрим!

Ты обязательно должен обратить внимание на то, что система координат в *Pygame*'е отличается от традиционной, в школе рассматриваемой системы. В школе рассматривается четыре четверти. В *Pygame*'е точка **(0, 0)** находится не в центре экрана, а **всегда в левом верхнем углу**. X-ось двигается вправо, а ось y опускается вниз. При выборе позиции для фигуры или картинки всегда надо делать отсчёт от левого верхнего угла. Порядок записи координат всё-таки стандартен - сначала значение по оси x, потом по оси y.

## Цвета

При работе с цветами используется **RGB**-система, которая представляет из себя комбинацию трёх чисел. Первое число показывает значение для красного (**R**ed) цвета, второе для зелёного (**G**reen) и последнее работает с концентрацией синего (**B**lue) цвета. Каждый цвет можно задать в промежутке от 0 до 255. Чем больше число, тем больше доля такого цвета в итоговой комбинации. Если все значения будут равны 0, то получим чёрный цвет, а если выбрать максимальные значения, т.е. 255, то перед нами будет белый цвет. Все другие цвета получим при использовании различных комбинаций. Если выставить одинаковые значения (например, [100, 100, 100]), то получим тона серого цвета. Чем выше значения одинаковых чисел, тем более светлый серый мы получим. Соответственно, чем меньше значения, тем более тёмным будет серый цвет.

На самом деле в *Pygame*'е можно использовать названия около 600 цветов, без использования *RGB*-системы. Примеры и названия (наведи мышкой на понравившейся вариант) приведены на веб-странице: <https://sites.google.com/site/meticulousslacker/pygame-thecolors>.

## Расположение различных фигур

Теперь поговорим об одном фокусе **pygame.draw**'а, который позволяет расположить фигуры относительно друг друга.

Предположим, что посередине экрана имеется прямоугольник или квадрат. Внутри него в самой середине должен располагаться нарисованный круг. Вычисляющие местоположения  $x$  и  $y$  для центра окружности в системе координат возьмёт время. На помощь приходит объект **pygame.draw** под названием **Rect**. В принципе это команда создания прямоугольника, но его можно использовать и отдельно, если в виде параметров задать координаты левого и верхнего угла, длину и ширину. Если такой объект определён, задан, то можно дальше использовать различные команды, чтобы создаваемые фигуры располагались каким-то образом относительно этого прямоугольника:

- относительно сторон: *top, left, bottom, right*
- относительно углов: *topleft, bottomleft, topright, bottomright*
- в середине краёв: *midtop, midleft, midbottom, midright*
- по середине: *center, centerx, centery*
- размер: *size, width, heigh*

**Изучи пример и попробуй самостоятельно выполнить**



```

import pygame, sys
pygame.init()
# установка размеров классического окна
окно = pygame.display.set_mode([640,480])
# заполняем белым цветом
окно.fill([255,255,255])

# определяем размеры коробочки типа rect
korobka = pygame.Rect (100,50,150,150)

# изображаем коробку на экране
pygame.draw.rect(окно, [0,255,0], korobka, 0)

# рисуем серый круг внутри квадрата,
# радиус 50
pygame.draw.circle(окно, [100,100,100], korobka.center, 50, 0)

# рисуем синюю линию на экране, стартовая позиция -
# центральная точка левого края квадрата, конечная
# точка находится в центральной точке экрана, толщина
# линии 3. Обрати внимание, что окно - это прямоугольник
# немного другого типа, то для использования rect'a
# необходимо сначала соединить окно с rect'ом, используя
# команду get_bounding_rect
pygame.draw.line(окно, [0,0,255], korobka.midleft, окно.get_bounding_rect().center, 3)

# Pygame сначала создаст экранную картинку и с помощью
# команды flip() обновит весь экран за один раз
pygame.display.flip()

while True:
    for i in pygame.event.get():
        if i.type == pygame.QUIT:
            sys.exit ()

# Цикл длится бесконечно, т.к. True всегда будет True.
# Внутри цикла вторым циклом проверяется последовательность
# событий Pygame'a. Если тип события совпадает с ним (т.е.
# нажали кнопку выхода (QUIT), то вся система закрывается

```

**NB!** Очень важно понимать, что **Rect** может быть задан любого размера и в любом месте, но при этом **не обязательно** выводить этот прямоугольник реально на экран. Таким образом, можно на экране использовать несколько коробок и коробочек, что позволит в дальнейшем легче ориентироваться при выборе позиции для других фигур. Вот такой фокус.



Посмотри видео.

## Рамки для фигур

Последним параметром при рисовании фигуры является толщина рамки. Во всех предыдущих примерах толщина была равна 0. Это означает, что фигуру на самом деле не рисовали, а заполняли целиком каким-то цветом. Если изменить значение на отличное от 0, то рамка

будет нарисована, а внутри будет пусто (что-то типо бублика). Толщина линии определяется значением цифры.

## Пузырьки

Разберись с приведённым ниже кодом самостоятельно.



```
import pygame, sys, random
pygame.init()
okno = pygame.display.set_mode([640,480])
okno.fill([255,255,255])

for i in range(100):
    x = random.randint(0,600)
    y = random.randint(0,450)
    radius = random.randint(5,60)
    ramka = random.randint(1,5)
    R = random.randint(0,255)
    G = random.randint(0,255)
    B = random.randint(0,255)
    pygame.draw.circle(okno, [R,G,B], [x,y], radius, ramka)
pygame.display.flip()

while True:
    for i in pygame.event.get():
        if i.type == pygame.QUIT:
            sys.exit()
```

Если запускать этот код несколько раз, то каждый раз будет выводиться новый результат (различного размера и цвета пузырьки).

## Произвольная линия

Помнишь, когда-то давно ты соединял на бумаге линии по пронумерованным точкам и получал в результате изображение? Создание произвольной линии в *Pygame*'е идёт так же. Сначала необходимо создать череду точек, которые необходимо с помощью линий соединить. Для этого будет использована команда **lines**, которая входит в **pygame.draw**. Последовательность точек обычно задают с помощью списка, потому что порядок соединения точек в таком списке уже определён.



## Проработай несколько примеров



```
import pygame, sys, random
pygame.init()
okno = pygame.display.set_mode([640,480])
okno.fill([255,255,255])

tochki = [(300,50), (350,100), (400,50), (400,150), (300,250)]

# линия чёрного цвета и толщиной 2 соединяет все точки в
# списке. True означает, что соединяется начальная и конечная
# точки.
pygame.draw.lines(okno,[0,0,0],True, tochki,2)

pygame.display.flip()

while True:
    for i in pygame.event.get():
        if i.type == pygame.QUIT:
            sys.exit ()
```

Для получения более плавной линии необходимо расположить точки очень плотно друг с другом. Чем больше будет записано точек в списке, тем более плавной получится линия. Однако выполнять такое в ручную очень затратно по времени. Поэтому для некоторых плавных линий используются обычные математические функции. Например, попробуем нарисовать волнистую линию с помощью синusoида:

```
import pygame, sys, math
pygame.init()
okno = pygame.display.set_mode([640,480])
okno.fill([255,255,255])

# определение пустого списка
tochki = []

for x in range(0,640): # возьмём по отдельности все координаты оси x
    y = int(math.sin(x/640.0*4*math.pi)*100+250)
    # координату для оси y определяем с помощью функции синуса,
    # чтобы волна получилась довольно большой, умножаем и складываем
    # значения
    tochki.append((x,y)) # сюда в список записывается 640 пар точек

# сейчас мы не хотим, чтобы начальная и конечная точки соединились,
# поэтому значение будет False.
pygame.draw.lines(okno,[0,0,0],False, tochki,1)

pygame.display.flip()

while True:
    for i in pygame.event.get():
        if i.type == pygame.QUIT:
            sys.exit ()
```



Посмотри видео.

## Текст

Выведение текстового сообщения на экран вполне обычная задача. Если ты изучал код игры Змейка внимательно, то для тебя не должно быть здесь никаких открытий. Текст можно вывести на экран следующим способом.

```
from pygame import *
init()
okno = display.set_mode([640,480])

# изображение текста указывается по умолчанию предложенным шрифтом (None), размер 40,
# местоположение по оси x будет 0, по оси y тоже 0. Команда render делает из
# расположенного между кавычками текста картинку, 1 означает, что у букв будут
# немного закруглённые края. Текст записан красным цветом [255,0,0]
okno.blit(font.Font(None,40).render('Текст 1 располагается в (0,0)',1,[255,0,0]),(0,0))

# тоже самое, что и предыдущий участок программного кода, только используется
# больше переменных
wriфт_текста = font.Font(None,30)
izobrazenije_текста2 = wriфт_текста.render('текст 2 располагается в (100,400)',1,[0,255,0])
okno.blit(izobrazenije_текста2,(100,400))

# расположим окно изображения по середине
izobrazenije_текста3 = wriфт_текста.render('текст 3 располагается по середине',1,[0,0,255])
рамка_текста3 = izobrazenije_текста3.get_rect()
рамка_текста3.center = okno.get_rect().center
okno.blit(izobrazenije_текста3,рамка_текста3)

display.flip()
while not event.get(QUIT):
    time.delay(10)
```

Имеются и другие шрифты, которые можно использовать вместо слова *None*. Их можно получить с помощью команды **pygame.font.get\_fonts()**. Запиши её в командную строку или в код с помощью команды *print()*. Изучи полученный таким образом список шрифтов.

## Картинки

Для добавления картинки на экран необходимо записать только две строчки кода. Сначала надо создать переменную, которая присваивается картинке, а затем вывести на экран значение созданной переменной.





Попробуем добавить вот такую птичку в окно будущей игры.

```
import pygame, sys

pygame.init()
okno = pygame.display.set_mode([640,480])
okno.fill([255,255,255])

# загружаем картинку, которая обязательно
# должна быть в том же каталоге, где и
# сам код
# местоположение птички в окне задано
# координатами 50 по оси x и 50 по оси y
ptichka = pygame.image.load("bird.png")
okno.blit(ptichka, (50,50))

pygame.display.flip()

while True:
    for i in pygame.event.get():
        if i.type == pygame.QUIT:
            sys.exit()
```

## Анимация

### Заставим картинки двигаться!

Перемещение картинок на мониторе означает не что иное, как рисование картинки в новом месте и "удаление" прошлой картинки со старого места. Удаление же на языке компьютера означает перекрашивание таким образом, что создаётся впечатление стёртой картинки. Если фон совпадает с цветом, то "удаление" происходит очень легко - помещаем над картинкой прямоугольник соответствующего цвета. Однако, если используется в виде фона какое-то изображение, текстура, то как быть?

Начнём с простого варианта, где фон будет белым и позволим объекту на экране передвигаться.



```

import pygame, sys

pygame.init()
okno = pygame.display.set_mode([640,480])
okno.fill([255,255,255])

# загружаем картинку, которая обязательно
# должна быть в том же каталоге, где и
# сам код
# местоположение птички в окне задано
# координатами 50 по оси x и 50 по оси y
ptichka = pygame.image.load("bird.png")
okno.blit(ptichka, (50,50))

pygame.display.flip()

# переместим птичку первый раз
pygame.time.delay(1000)
okno.blit(ptichka, (150,150))
pygame.draw.rect(okno, [255,255,255], [50,50,50,50],0)
pygame.display.flip()

# переместим птичку второй раз
pygame.time.delay(1000)
okno.blit(ptichka, (250,50))
pygame.draw.rect(okno, [255,255,255], [150,50,50,50],0)
pygame.display.flip()

while True:
    for i in pygame.event.get():
        if i.type == pygame.QUIT:
            sys.exit()

```

## Перемещение по прямой

В прошлой главе наша птичка просто перепрыгивала с одной точки в другую. Попробуем сделать это перемещение более плавным. Воспользуемся тем же методом, что и при создании плавной линии. Количество точек, где должна быть нарисована картинка, будет большим. Для совершения одной и той же процедуры много десятков раз - нарисуй и перекрась предыдущую картинку - будет использоваться цикл.

Изучим перемещение птички ещё раз, но с применением цикла.



```

import pygame, sys

pygame.init()
okno = pygame.display.set_mode([640,480])
okno.fill([255,255,255])

# значения для координат x и y будет 50
ptichka = pygame.image.load("bird.png")
x, y = 50,50

okno.blit(ptichka, (x,y))
pygame.display.flip()

for i in range(100):      # переместим птичку 100 раз
    pygame.time.delay(20)
    pygame.draw.rect(okno, [255,255,255], [x,y,50,50],0)
    x = x + 5             # изменим координату x на 5 единиц
    okno.blit(ptichka, (x,y))
    pygame.display.flip()

while True:
    for i in pygame.event.get():
        if i.type == pygame.QUIT:
            sys.exit()

```

Интересно, но намного приятнее видеть, когда птичка не летит по прямой линии, а, например, по синусоидальной траектории. Об этом уже в следующей главе.

## Перемещение по кривой

Для перемещения объекта по кривой необходимо воспользоваться математической функцией, которая описывает задуманную траекторию. С помощью неё предстоит довольно плотно рассчитать расположенные пары координат для точек, не забывая при этом увеличивать значение одной из координат.

## Запускаем птичку по траектории

```
import pygame, sys, math

pygame.init()
okno = pygame.display.set_mode([640,480])
okno.fill([255,255,255])

# значения для координат x и y будет 0 и 240, соответственно
ptichka = pygame.image.load("bird.png")
x, y = 0,240

okno.blit(ptichka, (x,y))
pygame.display.flip()

for i in range(100):    # переместим птичку 100 раз
    pygame.time.delay(40)
    pygame.draw.rect(okno, [255,255,255], [x,y,50,50],0)
    x = x + 5           # изменим координату x на 5 единиц
    y = int(math.sin(x/640.0*4*math.pi)*100+250)
    okno.blit(ptichka, (x,y))
    pygame.display.flip()

while True:
    for i in pygame.event.get():
        if i.type == pygame.QUIT:
            sys.exit()
```



Посмотри видео.

## Управление с клавиатуры

На данный момент наша птичка может летать как по прямой, так и вдоль кривой линий. Но хотелось бы самому определять местоположение птички. Как это сделать?

Для этого можно воспользоваться контролем событий **pygame.event.get()** и уточним у него, какую клавишу нажали (**pygame.KEYDOWN**). В зависимости от того, кнопка с какой стрелкой была нажата (**K\_UP**, **K\_DOWN**, **K\_LEFT**, **K\_RIGHT**), изменятся и координаты для рисования. На самом деле, можно проверить нажатие любой другой клавиши. Дополнительную информацию по работе **pygame.KEYDOWN** можно найти в документации по адресу <http://www.pygame.org/docs/ref/key.html>.

```

import pygame, sys

pygame.init()
okno = pygame.display.set_mode([640,480])

# значения для координат x будет 0, для координаты
# y это 240, размер шага 10 единиц
ptichka = pygame.image.load("bird.png")
x, y, wag = 0,240,10

while True:
    okno.fill([255,255,255])
    okno.blit(ptichka,(x,y))
    pygame.display.flip()
    pygame.time.delay(40)
    for i in pygame.event.get():
        if i.type == pygame.QUIT:
            sys.exit()
        elif i.type == pygame.KEYDOWN:
            if i.key == pygame.K_UP:
                y = y - wag
            elif i.key == pygame.K_DOWN:
                y = y + wag
            elif i.key == pygame.K_LEFT:
                x = x - wag
            elif i.key == pygame.K_RIGHT:
                x = x + wag

```

**NB!** Запуская данную программу, ты наверное обратил внимание, что сколько не держи нажатой клавишу направления, птичка всё равно перемещается на один шаг и то, только тогда, когда происходит непосредственное нажатие на клавишу. Для продолжения движения во время удерживания клавиши нажатой необходимо перед *while*-циклом записать ещё три строчки:

```
zaderzka = 100
```

```
interval = 50
```

```
pygame.key.set_repeat(zaderzka,interval)
```

## События, которые можно контролировать

В прошлой главе рассмотрели, как можно прописать в коде контроль над нажатием клавиши управления. На самом деле в *Pygame*'е довольно много событий, которыми можно управлять. Самые известные из них это:

- *KEYDOWN* - изучи в прошлой главе;
- *KEYUP*;
- *QUIT*;

- *MOUSEMOTION*;
- *MOUSEBUTTONUP*;
- *MOUSEBUTTONDOWN*.

Весь список событий, которые могут быть использованы для подключения игровой консоли или использования видео, приведены по адресу <http://www.pygame.org/docs/ref/event.html>.

Использование всех таких событий через **pygame.event** происходит таким же образом и способом, как мы использовали при работе с событием *KEYDOWN*.

## Управление мышкой

Перемещать объекты с помощью клавиатуры довольно легко. Необходимо "прослушивать" использование команд *KEYDOWN*, записать *if*-предложение о том, что необходимо в том или ином случае сделать. По такой же аналогии идёт и использование мышки для управления объектами по экрану. Для этого необходимо будет прослушивать событие **MOUSEMOTION** и записать программный код для реагирования на такое событие.

Далее на двух примерах рассмотрим перемещение объекта с помощью мышки. В первом случае объект будет перемещаться одновременно с мышкой, для работы со вторым примером необходимо будет держать нажатой кнопку мышки и тогда перемещать объект.

### Пример 1: Перемещение объекта вместе с мышкой

```
import pygame, sys

pygame.init()
okno = pygame.display.set_mode([640,480])
ptichka = pygame.image.load('bird.png')

okno.fill([255,255,255])
okno.blit(ptichka,(0,0))
pygame.display.flip()

while True:
    for i in pygame.event.get():
        if i.type == pygame.QUIT:
            sys.exit()
        elif i.type == pygame.MOUSEMOTION:
            okno.fill([255,255,255])
            okno.blit(ptichka, i.pos)
            pygame.display.flip()
```



## Пример 2: Объект перемещается при нажатой над ним кнопкой мышки

```
import pygame, sys

pygame.init()
okno = pygame.display.set_mode([640,480])
ptichka = pygame.image.load('bird.png')

# первоначальные координаты птички
x, y = 0,0
okno.fill([255,255,255])
okno.blit(ptichka, (x,y))
pygame.display.flip()
peremestit = False

while True:
    for i in pygame.event.get():
        if i.type == pygame.QUIT:
            sys.exit()
        elif i.type == pygame.MOUSEBUTTONDOWN:
            mx, my = i.pos
            if abs(mx-x)<50 and abs(my-y)<50:
                peremestit = True
                # двумя строчками проверяется, находится ли мышь над объектом
                # птичка. Если да, то разрешается перемещение.

        elif i.type == pygame.MOUSEBUTTONUP:
            peremestit = False
        elif peremestit and i.type == pygame.MOUSEMOTION:
            x,y = i.pos # передаются координаты мышки
            okno.fill([255,255,255])
            okno.blit(ptichka, (x,y))
            pygame.display.flip()

    # в этом elif'e есть разрешение на перемещение - проверяется,
    # а движется ли мышь и в соответствии с позицией мышки
    # происходит постоянное обновление картинки экрана
```



Посмотри видео.

## Финал игры Змейка

На этой неделе мы закончили работу над игрой Змейка. Конечно, можно ещё что-то дополнить, что-то придумать - было бы желание!

Приступи к изучению кода этой недели - что изменилось, что произошло? Попробуй запустить код на своём компьютере.

**NB!** Код можно посмотреть в отдельном [окне](#).

## Версия 8

**NB!** Код можно посмотреть в отдельном [окне](#).

## Версия 9

**NB!** Код можно посмотреть в отдельном [окне](#).

## Что ты узнал?

Вот и закончился материал ещё одной недели. Добавились новые кусочки кода, которые сможешь в дальнейшем применять. Итак, на этой неделе ты изучил и научился:

- использовать *Pygame*;
- создавать окно игры;
- рисовать в нём и добавлять в него фигуры;
- разбираться в рамках фигур;
- использовать цвета в *Python*'е;
- рисовать произвольную линию;
- измерять местоположение фигур относительно друг друга;
- добавлять и перемещать картинку на экране;
- обновлять картинку на экране;
- перемещать объекты по кривой;
- управлять объектами с помощью клавиатуры;
- держать и использовать контроль над событиями;
- перемещать объекты с помощью мышки;
- получил обзор о других контролируемых событиях в *Pygame*'е, которые могут пригодиться тебе в дальнейшем.





## НЕДЕЛЯ 6: ЗВУК И ЕЩЁ КОЕ-ЧТО ПОЛЕЗНОЕ

### Звук

Играть в игру, где не используется звук, довольно скучно. На примере игры Змейка мы уже работали со звуковыми файлами. Наличие музыкального сопровождения делает игру более увлекательной, приятной и неожиданной.

Звук может быть как вводной информацией, так и выводимой. Вводимая информация - это не что иное, как сохранение записи с помощью микрофона. Хотя такое применение в игровой индустрии практически не находит применения. А вот заниматься выводом звуковой информации мы будем на страницах данной книги. Рассмотрим небольшие звуковые эффекты и использование музыки как фона.

Использование звука опять довольно капризное мероприятие, надо знать особенности операционных систем, звуковых карт и т.д. Могут возникнуть такие же трудности, как и при работе с графикой. Но нашу работу облегчит *Pygame*, который позволяет воспользоваться `pygame.mixer`'ом.

### Звуковые файлы

Чтобы в программе зазвучала музыка, её необходимо сохранить на компьютере в виде файла. Такой файл можно создать самому, скачать из Всемирной паутины или взять с диска. Существует довольно много форматов и их расширений: `.wav`, `.mp3`, `.wma`, `.ogg` и др. Чаще всего используется первых два. Желательно размещать звуковые файлы в той же папке, где находится сам программный файл. Тогда при написании кода достаточно будет сослаться на название файла. В противном случае надо будет указать точный (и длинный) путь до файла. Например: `fonovaja_muzika` = `"C:\Program Files\Games\Music\back.wav"`.

### Pygame.mixer

Чтобы можно было использовать музыку, необходимо запустить `pygame.mixer`. Для этого в самом начале программного кода придётся прописать соответствующую команду (как и при работе с графикой). Она будет выглядеть следующим образом:

```
import pygame

pygame.init()

pygame.mixer.init()
```

Теперь все приготовления сделаны и можем приступить к запуску музыки. Это может быть небольшой музыкальный фрагмент (назовём его звуковым эффектом) или длинное произведение, которое будет играть фоном. **Pygame.mixer** обрабатывает их по-разному.

### Звуковые эффекты

Обычно они сохранены в формате *.wav* и для их запуска в *Pygame*'е необходимо воспользоваться командой **Sound**. *Sound* сразу загружает звуковой файл в память компьютера и оттуда происходит проигрывание звука:

```
zvuk_sjedennaja_miw = pygame.mixer.Sound(
    'soodud.wav')

zvuk_sjedennaja_miw.play()
```

### Длинное музыкальное сопровождение

Обычно фоновая музыка сохраняется как *.mp3*-файл. При использовании её не очень целесообразно загружать всю мелодию в память компьютера. В этом случае логичнее использовать возможность непосредственного проигрывания звука из файла и это делается с помощью команды *Pygame*'а **music.load()**:

```
pygame.mixer.music.load("BackgroundMusic.mp3")

pygame.mixer.music.play()
```

### Пример

Если программа должна только проиграть какой-то фрагмент, то разумнее отложить запуск проигрывания звука на небольшое количество времени. Это надо для того, чтобы *mixer* успел включиться в работу. Файл для примера можно скачать по адресу: <http://math.ut.ee/~kull/mustrastas.ogg>.



```
import pygame, sys
pygame.init()
pygame.mixer.init()
okno = pygame.display.set_mode([640,480])
pygame.time.delay(1000)

zvuk_ptichki = pygame.mixer.Sound('mustrastas.ogg')
zvuk_ptichki.play()

while True:
    for i in pygame.event.get():
        if i.type == pygame.QUIT:
            pygame.quit()
            sys.exit()
```

## Контроль за громкостью звука

Когда звук играет, то было бы очень удобно регулировать уровень громкости - либо потише, либо погромче. У *Pygame.mixer*'а для этих целей существует специальный отдельный метод **set\_volume()**. Для этого метода необходимо задать аргумент в виде числа, которое находится в промежутке от 0 до 1 (например, 0.3 или 0.8). Чем меньше значение, тем тише звучит музыка.

В следующем программном коде уровень звука отмечен на отметке 0.2, но во время изучения работы кода можешь использовать различные значения, чтобы понять всю прелесть использования метода.

```
import pygame, sys
pygame.init()
pygame.mixer.init()
okno = pygame.display.set_mode([640,480])
pygame.time.delay(1000)

zvuk_ptichki = pygame.mixer.Sound('mustrastas.ogg')
zvuk_ptichki.set_volume(0.2)
zvuk_ptichki.play()

while True:
    for i in pygame.event.get():
        if i.type == pygame.QUIT:
            pygame.quit()
            sys.exit()
```

## Фоновая музыка

Автор программного кода никогда не знает, как долго пользователь будет играть в его игру. Поэтому необходимо предусмотреть постоянное проигрывание музыки вне зависимости от продолжительности использования игры.

Решить такую проблему можно очень быстро и легко, добавив только одно число. Для этого аргументом функции **play()** надо задать любое негативное число - обычно это -1. Например, `zvuk_ptichki.play(-1)`.

Если же есть необходимость повторять музыкальное произведение какой-то конкретное количество раз (например, постучать три раза), то в функцию *play()* следует записать то число, сколько раз должна музыка проигрываться. Например, `zvuk_ptichki.play(3)`.

## Кое-что полезное

На следующих страницах ты можешь найти ещё немного материала, который может быть интересен тебе для создания игры. Как записывать счёт, оформлять столкновение и т.д. мы уже рассмотрели на страницах предыдущих книг. Здесь же найдёшь информацию о том, как создать таблицу первенства на основании результатов игроков, а также как спросить данный игрока.

Удачи в изучении хитростей!

## Таблица первенства - запись в файл

Приведённый ниже программный код генерирует случайным образом сумму пунктов, которую потом пользователь может поместить в турнирную таблицу. На экране будет отображено только три самых лучших результата. Каким образом?

Если турнирной таблицы ещё нет, то вначале такой файл пустой. В таком случае программа сама создаёт файл с турнирной таблицей и будет постепенно добавлять туда результаты. Если файл ранее уже существовал, то он открывается с таблицей, где уже записаны все предыдущие результаты. Далее проверяется, является ли новый результат лучше чем те, что уже были там записаны и добавляет его. Поиск нужного места в таблице основывается на работе со списками - данные можно отсортировать по пунктам. Если список отсортирован, то на экране отображается три лучших результата.

```

from random import randint
import pickle

razmer_tablici = 3
mesto_v_tablice = None
try:
    fail = open('tablica.pickle', 'rb')
    tablica = pickle.load(fail)
    fail.close()
except:
    tablica = []

def pechatat_tablicu():
    print('Место\tПункты\tИмя')
    for i in range(len(tablica)):
        (punkti, imja) = tablica[i]
        print(str(i+1)+'\t'+str(punkti)+'\t'+imja)

pechatat_tablicu()

punkti = randint(0,100)
print('Получил '+str(punkti)+' пунктов')

tablica.append((punkti, ''))
tablica.sort(reverse=True)
tablica = tablica[:razmer_tablici]
try:
    mesto_v_tablice = tablica.index((punkti, ''))
    print('Ты занял место №'+str(mesto_v_tablice+1))
    print('Введи своё имя:')
    imja = input()
    tablica[mesto_v_tablice] = (punkti, imja)
    print('Новая таблица:')
    pechatat_tablicu()
except:
    mesto_v_tablice = None
    print('К сожалению, ты не попал в турнирную таблицу')

fail = open('tablica.pickle', 'wb')
pickle.dump(tablica, fail)
fail.close()

```

## Узнать имя игрока

Данный программный код - это пример того, как в *Pygame*'е можно спросить имя игрока и вывести его затем на экране.

```

# -*- coding: utf-8 -*-

from pygame import *

init()
okno = display.set_mode([640,480])

cvet_fona = (255,255,255)
cvet_korobki = (255,0,0)
cvet_teksta = (0,0,255)
korobka = Rect((0,0),(360,120))
korobka.center = okno.get_rect().center
try:
    mesto_wrifta = font.match_font('chalkboard')
except:
    mesto_wrifta = None
imja_wrifta = font.Font(mesto_wrifta,50)
vvesti_wrift = font.Font(mesto_wrifta,30)
wrift_privetstviya = font.Font(mesto_wrifta,30)
foto_vvoda = vvesti_wrift.render('Введи, пожалуйста, своё имя латиницей:',1, cvet_teksta)
ramka_foto_vvoda = foto_vvoda.get_rect()
ramka_foto_vvoda.midbottom = korobka.midtop

def risovat():
    okno.fill(cvet_fona)
    okno.blit(foto_vvoda, ramka_foto_vvoda)
    draw.rect(okno, cvet_korobki, korobka, 3)
    foto_imja = imja_wrifta.render(imja, 1, cvet_teksta)
    foto_imja_raam = foto_imja.get_rect()
    foto_imja_raam.center = korobka.center
    okno.blit(foto_imja, foto_imja_raam)
    display.flip()

imja = ''
risovat()
while True:
    e = event.wait()
    if e.type == KEYDOWN:
        if e.key == K_BACKSPACE:
            imja = imja[:-1]
        elif e.key == K_RETURN:
            break
        elif e.key <= 127:
            if key.get_mods() & KMOD_SHIFT:
                imja += chr(e.key).upper()
            else:
                imja += chr(e.key)
        foto_imja = imja_wrifta.render(imja, 1, cvet_teksta)
        if foto_imja.get_rect().width > korobka.width:
            imja = imja[:-1]
        risovat()
    if e.type == QUIT:
        quit()
        exit()

wrift_privetstviya = wrift_privetstviya.render('Привет, '+imja+'!', 1, cvet_teksta)
ramka_wrifta_privetstviya = wrift_privetstviya.get_rect()
ramka_wrifta_privetstviya.midtop = korobka.midbottom
okno.blit(wrift_privetstviya, ramka_wrifta_privetstviya)
display.flip()

while not event.get(QUIT):
    time.delay(100)
quit()

```



## Итоговая игра Змейка

Твоя игра Змейка готова, но нет пределу совершенства. На странице <http://math.ut.ee/~kull/uss10mets/> ты можешь изучить материалы, коды, которые позволяют усовершенствовать данную игру. При желании изучи материал самостоятельно.

## Немного для саморазвития

Вот мы и подошли к самой последней части учебного материала. Если бы ты хотел углубить свои знания по программированию, то вот тебе несколько рекомендаций.

- В изучении темы игр тебе поможет бесплатная англоязычная интернет-книга. В ней приведены описания, примеры, что очень удобно для самостоятельного изучения. К сожалению, или к счастью, там не надо выполнять домашние задания, но и помощи нельзя спросить в форуме. Книга находится по веб-адресу <http://inventwithpython.com/chapters/>.
- Свои навыки в составлении игр можно совершенствовать, участвуя в тематических соревнованиях. У тебя сейчас уже имеется определённый багаж знаний, чтобы принимать участие в таких мероприятиях. Каждый год *Playtech* проводит конкурсы с призами. Дополнительная информация: <http://www.playtech.ee/?nav=news&news=264>.
- Можешь самостоятельно изучить материалы для первокурсников Тартуского университета по специальности Информатика. Например, здесь имеется материал по программированию для тех, кто ещё его не изучал: <http://courses.cs.ut.ee/2011/programmeerimine/Main/HomePage>. При желании загляни на <http://courses.cs.ut.ee/> - там тоже есть интересные материалы.
- Если ты уже заканчиваешь 12 класс, то осенью приходи учиться на специальность Информатика при Тартуском университете. Интересующую тебя информацию ты найдёшь на веб-странице института: <http://www.cs.ut.ee/>. Если будешь хорошо учиться, то у тебя будет очень перспективное будущее, высокооплачиваемая работа. Удачи!

## **ИСПОЛЬЗОВАННЫЙ МАТЕРИАЛ**

Автором оригинального курса является Тийна Кулл