

Python с нуля



**От новичка до собственных
игр и программ!**

**Roman
Gurbanov**

**Jean-Loup
Chrétien**

PYTHON С НУЛЯ

От новичка до собственных игр и программ

Спец издание к 41-летию миссии Союз Т-6

Авторы: Роман Гурбанов, Жан-Лу Кретьен
ISBN: 979-8-22385-304-6 | 2023

По всем вопросам: roman@qiber.org

ПРЕДИСЛОВИЕ	7
ВВЕДЕНИЕ	8
1. Как получить максимум от этой книги?	8
2. Куда записывать код?	10
3. Как читать код в этой книге?	10
4. Что делать с тестами из книги?	11
5. Почему именно Питон?	11
ГЛАВА ПЕРВАЯ: НАЧИНАЕМ ПРОГРАММИРОВАТЬ НА PYTHON!	12
1. Ваша первая строка кода	12
2. Что такое программа?	13
3. Функция Print	14
4. Как Python читает код?	15
5. Программа подсчета	15
6. Самостоятельная работа	15
7. Итоги первой главы	16
8. Тест первой главы	16
ГЛАВА ВТОРАЯ: ПЕРЕМЕННЫЕ	17
1. Что такое переменная?	17
2. Как создать и вывести переменную?	18
3. Итоги второй главы	19
4. Тест второй главы	19
ГЛАВА ТРЕТЬЯ: ЧИСЛА	20
1. Целые и дробные числа	20
2. Математические Операторы в Python	21
3. Работаем с числами	21
4. Делим числа без остатка в Python	22
5. Порядок вычислений в Python	22
6. Числа и переменные в Python	23
7. Итоги третьей главы	24
8. Тест третьей главы	24
ГЛАВА ЧЕТВЕРТАЯ: СТРОКИ	25
1. Строки в Python	25
2. Строки и функция печати	25
3. Хранение строк в переменных	26
4. Конкатенация строк в Python	27
5. Конкатенация строк и переменные	27
6. Форматирование строк в Python	28

7. Итоги четвертой главы	29
8. Тест четвертой главы	30
ГЛАВА ПЯТАЯ: БУЛЕВА ЛОГИКА	30
1. Операторы сравнения	31
2. Булевы значения: правда или ложь	31
3. True и False в переменных	33
4. Сравнение переменных в Python	33
5. Программа для проверки пароля	34
6. Итоги пятой главы	36
7. Тест пятой главы	36
ГЛАВА ШЕСТАЯ: УСЛОВНЫЕ ЗАЯВЛЕНИЯ	37
1. Что такое условные операторы?	37
2. Условный оператор If	38
2.1. if и операторы сравнения	39
2.2. Условный оператор if и числа	40
3. Условный оператор else	41
4. Условный оператор elif	42
5. Порядок if, elif и else	44
6. Итоги шестой главы	44
7. Тест шестой главы	45
ГЛАВА СЕДЬМАЯ: ЦИКЛЫ	46
1. Что такое Циклы?	46
2. Цикл while	48
3. Как остановить цикл while	48
4. Цикл while и операторы сравнения	49
5. Цикл while и обратный отсчет	50
6. Цикл for	52
7. Итоги седьмой главы	53
8. Тест седьмой главы	53
ГЛАВА ВОСЬМАЯ: СПИСКИ	54
1. Введение в списки	54
2. Индекс списка	55
3. Как извлечь значение из списка	56
4. Объединение значений списка	58
5. Функции списка	58
6. Функция len	59
7. Функция append	60

8. Объединение списков в Python	60
9. Кортежи	62
10. Как превратить кортеж в список?	63
11. Как превратить список в кортеж?	65
12. Итоги восьмой главы	65
13. Тест восьмой главы	66
ГЛАВА ДЕВЯТАЯ: СЛОВАРИ	67
1. Что такое словари?	67
2. Формат словаря в Python	67
3. Добавление пар в словарь	68
4. Удаление пар из словаря	69
5. Как получить значение по ключу?	70
6. Как получить ключ по значению?	70
7. Проверка ключа в словаре	71
8. Как проверить длину словаря?	71
9. Итоги девятой главы	72
10. Тест девятой главы	72
ГЛАВА ДЕСЯТАЯ: ФУНКЦИИ	73
1. Введение в функции	74
2. Параметры и аргументы функций	75
3. Как вернуть значение из функции	77
4. Вложенные функции	78
5. Итоги десятой главы	79
6. Тест десятой главы	80
ГЛАВА ОДИННАДЦАТАЯ: КЛАССЫ И ОБЪЕКТЫ	81
1. Введение в классы и объекты	81
2. Как создать класс в Python?	82
3. Свойства класса	82
4. Функции класса в Python	83
5. Как создать объект класса в Python	87
6. Управление объектами класса	92
7. Итоги одиннадцатой главы	95
8. Тест одиннадцатой главы	95
ГЛАВА ДВЕНАДЦАТАЯ: МОДУЛИ И ПАКЕТЫ	96
1. Введение в модули и пакеты	97
2. Как импортировать модуль	97
3. Как импортировать функцию модуля	100

4. Имя функции модуля в Python	101
5. Как создать модуль в Python	102
6. Пакет модулей в Python	102
7. Итоги двенадцатой главы	102
8. Тест двенадцатой главы	103
ГЛАВА ТРИНАДЦАТАЯ: ФИНАЛЬНЫЙ ПРОЕКТ	104
Код игры	105
Шаг 1/7: Введение в игру	107
Шаг 2/7: Вспоминаем классы и объекты в Python	108
Шаг 3/7: Детально разбираем атрибуты класса SoyuzDocking	109
Шаг 4/7: Детально разбираем методы класса SoyuzDocking	110
Шаг 5/7: Углубляемся в работу игрового цикла	111
Шаг 6/7: Учимся принимать и обрабатывать ответ игрока	113
Шаг 7/7: Учимся выводить сообщения для игрока	114
ГЛАВА ЧЕТЫРНАДЦАТАЯ: ЧТО ДАЛЬШЕ?	116
Разработка игр	117
Веб-приложения	117
Чат-боты	117
Кибербезопасность и тестирование на проникновение	117
Приложения машинного обучения	117
Научные и числовые приложения	118
Веб-скрапинг приложения	118
Автоматизация и написание скриптов	118
Блокчейн-приложения	118
Обработка изображений	118
Обработка естественного языка (NLP)	118
Интернет вещей (IoT)	118
Сетевое программирование	118
ПРИЛОЖЕНИЕ: ОТВЕТЫ К ТЕСТАМ	119
Тест первой главы	119
Тест второй главы	120
Тест третьей главы	121
Тест четвертой главы	122
Тест пятой главы	123
Тест шестой главы	124
Тест седьмой главы	126
Тест восьмой главы	128

Тест девятой главы	129
Тест десятой главы	131
Тест одиннадцатой главы	132
Тест двенадцатой главы	134

ПРЕДИСЛОВИЕ

Сегодня в ваших телефонах больше вычислительной мощности, чем в компьютерах космических кораблей, на которых я совершал полеты в космос.

Если этих компьютеров и программ хватало для покорения космоса, только представьте, что можете сделать вы, написав свои программы сегодня.

Высокие технологии делают нашу жизнь лучше, интереснее и безопаснее. Каждый, кто желает заниматься чем-то интересным и полезным в современном мире, должен с ними дружить. И мне кажется, изучение программирования по книгам, таким как эта, — один из лучших способов это сделать.

Жан-Лу Кретьен

Первый европеец, вышедший в открытый космос, астронавт NASA, Герой Советского Союза.

ВВЕДЕНИЕ

1. Как получить максимум от этой книги?

В этой книге четырнадцать глав. Двенадцать из них посвящены основам программирования на Python. А тринадцатая содержит Ваш финальный проект (и о нем чуть позже)

Если Вы пройдете все тринадцать глав, то получите крепкие базовые навыки в программировании на Python. Научитесь писать не сложный код, и подготовитесь к финальному проекту, который ждет Вас в конце этой книги.

В этом проекте, Вы создадите собственную программу для бортового компьютера космического корабля. И эта программа отвечает за стыковку корабля с космической станцией.

Стало интересно? :) Тогда продолжим!

Главы этой книги расположены по мере роста сложности: от простого к более сложному.

Если Вы хотите пропустить какую-то главу, Вы, конечно можете это сделать. Но Вы должны знать, что каждая такая глава содержит информацию, необходимую для прохождения следующей главы.

Поэтому советую идти по порядку, и ничего не пропускать :)

Все четырнадцать глав можно пройти залпом за несколько часов. Но, очень рекомендую разделить обучение на ежедневные, небольшие уроки. Хотя бы по 15–30 минут в день.

Если Вы действительно хотите научиться программировать на Python, то лучше учиться этому ежедневно, и понемногу. Чем раз в неделю с утра и до вечера.

Есть еще кое-что. В каждой главе есть готовый код для примера. Я очень рекомендую экспериментировать и создавать свои версии кода, как только вы освоите эти примеры.

Просто меняйте код и наблюдайте за тем, как он влияет на результат работы программы. Так Вы быстрее научитесь программировать.

Примеры кода будут частично на русском. А позже, когда мы перейдем к изучению классов и объектов, уже полностью на английском.

Не бойтесь, знания английского для этого курса вам не нужны.

Но, для того чтобы быть полноценным программистом, в будущем Вам будет необходимо владеть английским, хотя бы базовым.

Вы ведь хотите работать в Google? Эммм? Не хотите? Но знать английский все равно придется.

2. Куда записывать код?

Работая по этой книге, Вам не нужно устанавливать и настраивать никаких редакторов кода.

Просто заходите на PythonOnline.kz. Там Вы можете писать, запускать, проверять и даже скачивать Ваш код на компьютер. Используя встроенный компилятор.

Компилятор PythonOnline.kz был создан для того, чтобы Вы не заморачивались на установке и настройке редакторов кода, а могли сразу приступить к программированию.

В этой книге я могу называть этот компилятор разными словами: редактор, консоль, компилятор. Это все одно и то же.

3. Как читать код в этой книге?

Читая код в этой книге, пожалуйста обратите внимание на несколько нюансов:

Точки в начале некоторых строк кода указывают на отступы. Эти точки только в книге, и нужны они только для корректного отображения отступов. Вам не нужно ставить ни точки, ни отступы в своем коде, так как наш компилятор будет ставить отступы автоматически.

Между строчками кода стоят пробелы. Опять же, это только в книге. Вам не нужно делать пробелы между строчками в своем коде.

Но, Вы можете ставить пробелы между блоками кода.

Все, что стоит в коде за знаком `#` — это комментарии к коду, для Вашего удобства.

Если Вы сейчас что-то не поняли, не волуйтесь. Время от времени, я буду напоминать Вам об этих нюансах, по мере прохождения данной книги.

4. Что делать с тестами из книги?

Очень рекомендую проходить тест в конце каждой главы.

Отвечайте на вопросы в уме. В случае, если в тесте нужно работать с кодом, записывайте, запускайте и проверяйте его на [PythonOnline.kz](https://pythononline.kz)

Если что-то не получается с тестами, Вы всегда можете подсмотреть готовые ответы. Они есть в приложении, в конце книги.

Только не бегите за подсказками сразу! Лучше пройдите тему еще раз. А затем вернитесь к тесту, и перепройдите его.

Здесь нет строгих учителей. И перепроходить тесты можно сколько угодно, пока Вы не останетесь довольны своим результатом.

5. Почему именно Питон?

Python — один из самых простых в освоении, но в то же время один из самых популярных и широко используемых языков программирования во всем мире.

Я бы порекомендовал Python как первый язык программирования всем, кто хочет научиться программировать.

Почему?

У Python чистый, минималистичный синтаксис. И это упрощает написание и чтение кода.

Например, для того, чтобы написать небольшую программу на Python, Вам может потребоваться всего несколько строчек кода.

А для того, чтобы написать такую же программу, скажем, на Java или C++, Вам придется писать куда больше кода.

Именно поэтому, на технических собеседованиях в Google или еще куда, Вам позволят решать задачи на Python. Даже если Вас рассматривают на позицию разработчика Java или C++ и тд.

Python — это высокоуровневый язык программирования. Это означает то, что он автоматизирует многие процессы, такие, как управление памятью.

А это, в свою очередь, поможет Вам сосредоточиться на основных задачах, пока Вы пишете код.

Python чрезвычайно популярен в реальном мире. Возьмите для примера IT-гигантов, таких, как: Google, Apple, Netflix. Все они используют Python в повседневных задачах, связанных с обработкой данных, работой нейронных сетей, и других важными для этих компаний процессов.

Python, действительно, универсален. Он отлично работает не только для математических задач, связанных с данными. Но и для веб-приложений, видеоигр, и, вообще, чего угодно. И это благодаря огромному выбору расширений и библиотек, доступных для Python. Об этом Вы узнаете более подробно в четырнадцатой главе.

Наконец, у Python есть преданное и постоянно растущее сообщество разработчиков. Это означает, что количество сфер и задач, где применяется Python, будет только расти.

Ну, а про зарплаты программистов на Python, их карьерные возможности и спрос, я просто промолчу. Об этом уже итак с каждого угла крикнули :)

На этом все. Желаю Вам приятного обучения, и увидимся в следующей главе!

ГЛАВА ПЕРВАЯ: НАЧИНАЕМ ПРОГРАММИРОВАТЬ НА PYTHON!

1. Ваша первая строка кода

Любая, даже самая продвинутая программа на Python, начинается с первой строки кода.

Вот пример простой программы, которая состоит всего из одной строки. Все, что она делает, это выводит сообщение: “Привет! Это моя первая строка кода!”

Впишите эту строчку в компилятор, и запустите код:

```
print("Привет! Это моя первая строчка кода!")
```

Если Вы все сделали правильно, то компилятор ответит Вам вот таким сообщением:

“Привет! Это моя первая строка кода!”

Получилось? Поздравляю с первой строчкой кода!

И даже если Вы пока не понимаете что к чему, не волнуйтесь. Мы все разберем и узнаем по ходу прохождения книги. А пока продолжим!

2. Что такое программа?

Даже если Вы написали всего одну строчку кода, как в предыдущем разделе, Вы написали программу.

Такую же программу, как и те, что запускаются с компьютера или смартфона.

Но, что такое программа? Программа — это набор инструкций и правил для компьютера, написанный на языке программирования.

Если с этим все понятно, давайте продолжим и закрепим то, что мы пока узнали.

Вот код, который я перемешал:

"Илон Маск ест пингвинов!" () print

Впишите его в компилятор так, чтобы он заработал. Выполнив прошлую задачу, Вы сможете выполнить и эту.

Если Вы все сделали правильно, то вот, что вам ответит программа:

“Илон Маск ест пингвинов!”

Справились с задачей? Отлично!

В обеих программах, которые мы только что создали, мы использовали функцию print (печать).

В этой книге мы часто будем ее использовать. Но, для начала, давайте познакомимся с ней поближе.

3. Функция Print

Функция print делает именно то, чем она и называется. Она “печатает” текст на экране.

Программисты используют эту функцию для того, чтобы показывать сообщения пользователям программы.

Например, «Пожалуйста, войдите в систему, используя пароль» или «Ваш пароль слишком слабый, используйте более надежный пароль» и т. д.

Функция print выводит не только текст, но и результаты вычислений, представленные в виде цифр и чисел.

Об этом мы поговорим немного позже. А пока, давайте еще немного потренируемся с выводом текста.

Вот вам текст:

Эй! Я продолжаю кодить!

Впишите его в компилятор вместе с функцией print, так, чтобы программа вывела такое же сообщение на экран.

Вот, что у вас должно получиться:

“Эй! Я продолжаю кодить!”

Все получилось? Отлично! Теперь, когда Вы умеете выводить сообщения на экран, используя функцию `print`, Вы можете поэкспериментировать и повеселиться со своими сообщениями.

Придумайте что угодно, любые сообщения. Добавьте к сообщениям цифры и числа.

А еще, попробуйте добавить или удалить что-нибудь из своего кода. И посмотрите, что произойдет. Это лучший способ узнать, как работает Ваш код.

4. Как Python читает код?

Как только Вы запускаете код, компьютер начинает читать его построчно, сверху вниз. Точно также, как Вы сейчас читаете эту книгу.

Это может показаться не важным, но это следует учитывать при написании и организации нашего кода.

Вот почему некоторые элементы, такие как модули (мы изучим их позже), находятся в верхней части кода.

Мы всегда импортируем их сверху нашего основного кода. А затем вызываем эти модули, спускаясь ниже, строка за строкой.

5. Программа подсчета

Давайте создадим программу, которая считает от 1 до 3. Вот код, который нам для этого нужен:

```
print("1")  
print("2")  
print("3")
```

Впишите этот код в компилятор и запустите его.

Программа сосчитала до трех? Довольно просто, правда? Теперь

расширьте код, чтобы программа смогла сосчитать до 10.

Получилось? Программа считает до десяти? Отличная работа!

6. Самостоятельная работа

Давайте отметим завершение первой главы небольшим испытанием.

Все, что Вам нужно сделать, это вывести это сообщение на экран:

Программировать на Python легко!

Ну как? Смогли? Я и не сомневался в Вас. Программировать на Python действительно не сложно. А теперь к итогам!

7. Итоги первой главы

В первой главе Вы выполнили следующее:

1. Узнали, что такое Python;
2. Узнали, что такое программа;
3. Поняли, как Python читает код;
4. Написали первую строчку кода;
5. Создали несколько простых программ;
6. Научились использовать функцию `print`.

Я Вас поздравляю! Теперь Вы готовы ко второй главе — Переменные. Давайте приступим к ней!

8. Тест первой главы

1. Компьютерная программа — это:

1. Набор инструкций и правил для компьютера, написанный на языке программирования.
2. Кусок кода, написанный на компьютере.
3. Загружаемая игра.

2. Как сделать так, чтобы компьютер вывел сообщение на экран?

1. Используя волшебное слово.
2. Используя функцию `print()`.
3. Используя команду «Показать сообщение!»

3. В каком порядке компьютер обрабатывает (считывает) код?

1. Компьютер считывает код построчно. Сверху вниз.
2. Компьютер считывает код построчно. Снизу вверх.
3. Компьютер ничего не считывает; Он все помнит наизусть.

4. Расположите фрагменты кода так, чтобы программа отображала сообщение «Я люблю Python!»

1.)
2. (
3. "Я люблю Python!"
4. `print`

ГЛАВА ВТОРАЯ: ПЕРЕМЕННЫЕ

1. Что такое переменная?

Переменные в Python создаются просто. И в этой главе Вы с легкостью научитесь создавать и применять их в своем коде.

Итак, приступим.

Переменная — это простой тип данных, у которого есть имя и значение. Переменные нужны для того, чтобы хранить в них информацию.

Давайте объясню на примере.

Представьте себе машину. В нашем случае машина является переменной. Название переменной — машина.

У машины есть марка — Tesla. Это значение переменной.

Таким образом, переменная (машина), хранит информацию о марке машины — Tesla.

Вот, как эта переменная выглядит на языке Python:

```
машина = "Tesla"
```

Давайте разберем все по порядку:

Сначала мы дали нашей переменной имя — машина.

Затем выставили знак равенства =

И наконец, мы присвоили нашей переменной значение “Tesla”

Значение переменной всегда ставится в кавычки, как в нашем примере. Иначе переменная работать не будет.

Теперь, когда мы поняли, что такое переменная, и как ее прописать, давайте создадим нашу первую переменную, и выведем ее на экран!

2. Как создать и вывести переменную?

Для начала давайте запишем нашу переменную из примера выше в компилятор и запустим его:

```
машина = "Tesla"
```

Ничего не произошло? Все верно. Ведь переменная — это не сообщение, а всего лишь тип данных.

Для того, чтобы вывести переменную на экран, нам придется использовать функцию print, с которой Вы уже знакомы.

Вот, как мы это сделаем. Запишите следующий код в компилятор и запустите его:

```
машина = "Tesla"  
print(машина)
```

Если Вы все сделали правильно, компилятор вернет вам значение переменной — Tesla.

А теперь давайте разберем код по порядку:

Сначала мы объявили переменную и дали ей имя — машина.

Затем мы присвоили переменной значение — “Tesla”.

Затем на второй строке мы прописали функцию print, и передали в эту функцию имя нашей переменной, поместив его в скобки функции.

Каждый раз, когда мы создаем переменную, и передаем ее имя функции print, эта функция будет выводить значение переменной на экран, как в нашем примере.

Кстати, если имя Вашей переменной состоит из более, чем одного слова, тогда Вам необходимо соединить эти слова нижним подчеркиванием. Например: моя_машина.

Однако, советую называть Ваши переменные только одним словом. Это обычная и корректная практика в работе с переменными.

Ну вот, теперь Вы знаете, как программисты создают и выводят переменные в Python на экран.

Совсем не сложно, правда?

А теперь давайте немного потренируемся. Ниже приведен код, в котором кое-чего не хватает. Вам надо это исправить так, чтобы программа могла создать переменную и отобразить ее значение.

```
= " "  
( )
```

К этому моменту у Вас должно быть достаточно знаний и навыков для выполнения этого легкого задания.

Как только Вы справитесь с этим заданием, потренируйтесь еще немного. Вы можете изменить имя и значение переменной. Добавить и вывести больше новых переменных на экран.

Чем больше повторений Вы выполните, тем лучше закрепите и отточите полученные навыки!

3. Итоги второй главы

Во второй главе Вы выполнили следующее:

1. Узнали, что такое переменные;
2. Научились создавать переменные в Python;
3. Научились выводить значения переменных на экран.

Отличная работа! Переходим к третьей главе — Числа. Обещаю, никакой скучной математики! Приступим.

4. Тест второй главы

1. Для чего нужны переменные?

1. Переменные нужны для хранения информации.
2. Переменные нужны для изменения информации.
3. Переменные нужны для извлечения или удаления информации.

2. Если имя переменной состоит из двух и более слов, Вы должны соединить их с помощью:

1. Нижнего подчеркивания.
2. Пунктирной линии.
3. Никак, это нормально — слепить все слова в одно.
4. Нужно прописать слова слитно, с большой буквы.

3. Мы можем вывести переменную на экран следующим образом:

1. Используя функцию print и поместив команду print в скобки.
2. Используя функцию print и поместив значение переменной в скобки.
3. Используя функцию print и поместив имя переменной в скобки.

4. Расположите фрагменты кода в правильной последовательности, чтобы получилась переменная, которая выводит "Илон" на экран.

1. (имя)
2. имя
3. =

- 4. print
- 5. "Илон"

ГЛАВА ТРЕТЬЯ: ЧИСЛА

1. Целые и дробные числа

Числа в Python, как и в обычной школьной математике бывают целые и дробные (их еще называют вещественными)

И если Вы не часто прогуливали уроки, то уверен, Вы знакомы с целыми числами, например, 5 или 10. А еще Вы знакомы с дробными числами, такими как 5.5 или 10.7, верно?

В программировании целые числа называют integer, а дробные — float.

Ну так вот, Python прекрасно работает и с целыми, и с дробными числами.

А еще Python отлично работает с математическими операторами. Давайте вместе на них посмотрим.

2. Математические Операторы в Python

Python прекрасно справляется с вычислениями. Для этого он применяет вот такие математические операторы:

Сложение: +
Вычитание: —
Умножение: *
Деление: /

Давайте теперь немного поработаем с этими операторами и числами.

3. Работаем с числами

Для начала давайте создадим переменную под названием результат, и присвоим ей значение 2+2.

Затем выведем результат на экран. Вот как это будет выглядеть:

```
результат = 2+2  
print(результат)
```

Запишите этот код в компилятор, и запустите его.

Что получилось? Верно, получилось четыре.

А теперь, используя тот же код, вычтите 5 из 10, используя оператор вычитания.

Запускайте код.

Пятерка есть? Отлично.

Теперь давайте умножим 5 на 5, используя оператор умножения.

Двадцать пять получилось? Прекрасно.

Наконец, давайте разделим 10 на 2, используя оператор деления.

Что получилось? 5.0? Верно. Вы только что выполнили деление с остатком. Поэтому в результате у нас вышло дробное число.

А теперь давайте поговорим о делении без остатка.

4. Делим числа без остатка в Python

Итак, в прошлом примере у нас получилось дробное число (float).

Но, что если нам нужно получить целое число (integer)?

Это довольно просто. Для того, чтобы получить целое число при делении, все, что нам нужно сделать, это применить двойной оператор деления — //

Попробуйте сами, замените оператор деления на двойной оператор деления в нашем предыдущем примере, и запустите код:

```
результат = 10//2  
print(результат)
```

Если Вы все сделали правильно, то увидите integer равный 5

5. Порядок вычислений в Python

Python делает вычисления точно в таком же порядке, какому Вас учили в школе.

Посчитайте в уме вот такой пример:

$(5+5)*3$

А затем впишите его в компилятор и запустите код:

```
результат = (5+5)*3  
print(результат)
```

Вот как Python будет его решать:

Сначала Python вычислит все, что находится в скобках. Сделает он это в таком порядке: сначала умножение, затем деление, затем сложение, затем вычитание.

После этого, Python вычислит все, что находится за скобками. Сделает он это в том же порядке, что и выше (умножение, деление, сложение, вычитание).

Следовательно, Python сложит 5 и 5, что даст 10. И умножит 10 на 3, что даст 30.

Ну как, совпали Ваши результаты?

Хорошо! Тогда идем дальше.

6. Числа и переменные в Python

Как Вы уже заметили, работая с числами и математическими операторами в Python, мы также использовали переменные и функцию print.

Заметили, да?

Так вот, давайте теперь закрепим то, что мы сделали:

Во-первых, мы объявили переменную, дав ей имя «результат» и значение «(5+5)*3»;

Затем мы спустились на одну строку вниз, прописали функцию print, и передали ей имя нашей переменной;

Когда мы запустили код, Python вычислил (5+5)*3, получил 30, и присвоил это значение переменной «результат»;

Наконец, Python увидел функцию print с аргументом (результат), и понял, что надо вывести на экран значение переменной результат, которое как мы уже поняли равно 30.

Как видите, Python отлично комбинирует числа, математические операторы, переменные и функции.

А теперь давайте еще немного попрактикуемся и создадим свои примеры с числами и переменными, которые похожи на те, что мы только что использовали.

Вот несколько шаблонных примеров для Вас:

```
результат = 2+2  
print(результат)
```

```
результат = 10-5  
print(результат)
```

```
результат = 5*5  
print(результат)
```

```
результат = 10/2  
print(результат)
```

7. Итоги третьей главы

В третьей главе Вы сделали следующее:

1. Узнали об integer и float — целых и дробный числа;
2. Применили математические операторы в вычислениях в Python;
3. Научились делить число без остатка;
4. Узнали порядок вычислений;

5. Научились комбинировать числа, переменные и функции в Python.

Молодцы! С числами мы разобрались. Настало время научиться создавать и применять строки в Python.

8. Тест третьей главы

1. Какие типы чисел есть в Python?

1. В Python есть три типа чисел: целые, почти целые, и дробные числа.
2. В Python есть два типа чисел: целые и полуцелые.
3. В Python есть два типа чисел: Целые и дробные числа.

2. Как Python делит 10 на 5? (множественный выбор)

1. 10:5
2. 10/5
3. 10—5
4. 10 * 5
5. 10//5

3. Расположите фрагменты кода так, чтобы получилось целое число 4.

1. //
2. 10*2
3. 5

Вопрос 4: Этот код выводит число 5 на экран. Но кое чего в нем не хватает. Определите, чего именно?

```
результат = 10/2  
(результат)
```

ГЛАВА ЧЕТВЕРТАЯ: СТРОКИ

1. Строки в Python

Строки в Python — штука необходимая, и вот почему:

Помните сообщения, которые мы выводили в первой главе,

например: “Привет! Это моя первая строка кода!”

Так вот, это все строки.

В Python, строка — это тип данных, имеющий текстовый формат, или просто текст, заключенный в кавычки.

А значит, чтобы создать строку, мы должны ввести текст и заключить его в кавычки.

2. Строки и функция печати

Давайте напишем код, который бы отображал вот такую строку: “Теперь я знаю, что такое строка в Python.”

Вы уже знаете, как выводить строки на экран, в Python, верно?

Вот Вам код для начала. Введите этот код в компилятор, и запустите его:

```
print("Теперь я знаю, что такое строка в Python.")
```

А теперь, Ваша очередь. Придумайте какую-нибудь другую строку, перепишите и запустите приведенный выше код.

3. Хранение строк в переменных

Помните нашу первую переменную:

```
машина = "Tesla"  
print(машина)
```

Вот теперь Вы знаете, что значение переменной — “Tesla” имело формат строки, потому что мы заключали это значение в кавычки.

Давайте теперь закрепим наши знания вот таким небольшим заданием:

Мы создадим сообщение в формате строки. Сохраним его в переменной и отобразим значение переменной на экране с помощью функции print.

Давайте я сделаю это первым. Впишите этот код в компилятор и

запустите его:

```
сообщение = "Хьюстон, у нас проблема"  
print(сообщение)
```

Получилось вывести сообщение? Отлично!

Теперь Ваша очередь. Глядя на пример выше, придумайте свой вариант сообщения, который можно сохранить в переменную. А затем вывести значение переменной на экран.

Придумайте и запустите столько примеров, сколько пожелаете. Чем больше, тем лучше. Так Вы укрепите полученный навык!

4. Конкатенация строк в Python

Мы можем объединять различные строки друг с другом. Этот прием называется “конкатенация”.

Все, что нам нужно сделать, чтобы объединить (конкатенировать) строки, это поставить между ними оператор сложения +.

Ничего сложного, правда? Давайте попрактикуемся с конкатенацией:

```
"Илон Маск отправил Теслу на" + "Марс"
```

А теперь выведем обе строки на экран. Впишите этот код в компилятор и запустите его:

```
print("Илон Маск отправил теслу на" + "Марс")
```

Заметили нечто странное? Кажется, наши строки слиплись.

Не вопрос! Мы можем это легко исправить. Есть несколько способов. Вот самый простой:

Все, что нам нужно сделать, это оставить пробел между первой кавычкой второй строки и словом, которое идет за этой кавычкой:

```
print("Илон Маск отправил теслу на" + " Марс")
```

Исправили? Запускайте код.

5. Конкатенация строк и переменные

Хочу обратить Ваше внимание на то, что мы можем конкатенировать строку только с другой строкой. Или с другим значением, имеющим формат строки.

Например, если мы создали переменную, и присвоили ей значение в формате строки, то мы можем объединить такую переменную с другой строкой.

В приведенном ниже примере я создал переменную (марка), и присвоил ей строковое значение "Tesla".

Затем я вывел это значение в конкатенации с другой строкой "Машина называется".

Вот, что получилось:

```
марка = "Tesla"  
print("Машина называется " + марка)
```

Впишите этот код в компилятор и запустите его. Если Вы все сделали правильно, программа вернет сообщение "Машина называется Tesla"

А теперь потренируйтесь. Измените код по Вашему желанию. Вы даже можете объединить более двух строк!

Меняйте код и запускайте его. Наблюдайте за тем, как меняется результат.

6. Форматирование строк в Python

Мы уже научились объединять строки с помощью математического оператора +. Этот оператор может только конкатенировать строку с другой строкой.

Но что, если мы хотим конкатенировать строку с чем-то, что не имеет формата строки?

Для этого есть отличный способ! И он называется

“Форматирование строки”. Программисты часто им пользуются.

Давайте объединим строку с переменной. Для этого мы переведем значение переменной в формат строки.

Для этого нам понадобятся две вещи:

Первое — это метод `format()` для форматирования не строкового значения и вложения его внутрь строки-заполнителя.

Второе — это сам заполнитель — `{}` для не строкового значения.

Давайте я покажу Вам, как это работает, на примере ниже:

```
print("Меня зовут Джо, и мне {} лет".format(20))
```

Введите этот код в компилятор и запустите его. Если Вы все сделали правильно, программа вернет Вам сообщение: Меня зовут Джо, и мне 20 лет.

Получилось? Отлично. А теперь давайте разберем все по порядку:

Мы вставили заполнитель в нашу строку. Вы можете распознать этот заполнитель по фигурным скобкам — `{}`.

Этот заполнитель нужен для того, чтобы хранить в себе место для возраста нашего Джо, который имеет числовое значение.

В конце строки мы помещаем метод `format()`, и передаем в его скобки сам возраст — 20, который имеет числовое значение.

В результате этих нехитрых действий Python взял наше числовое значение, отформатировал его в строку, и поместил в заполнитель.

И, наконец, превратив числовой формат в строку, мы вывели всю строку на экран при помощи функции `print`.

Вот и все. Ничего сложного, верно?

Очень рекомендую Вам поиграть с этим кодом. Поэкспериментировать с заполнителями и значениями, которые Вы передаете методу `format()`.

Вот Вам более сложный пример, с двумя заполнителями:

```
print("Меня зовут {}, и мне {} лет".format("Джо",20))
```

Впишите этот пример в компилятор, и запустите его.

Потренируйтесь, объясните себе, как он работает. А затем придумайте свой вариант с двумя или более заполнителями.

7. Итоги четвертой главы

В четвертой главе Вы сделали следующее:

1. Узнали, что такое строки в Python;
2. Узнали, как хранить строки в переменной;
3. Научились конкатенировать строки в Python;
4. Научились конкатенировать строки со значениями переменных;
5. Узнали, что такое форматирование строки, и как переводить типы данных в формат строки.

Отличная работа! А впереди нас ждет очень интересная тема — Булева логика.

Приступим!

8. Тест четвертой главы

Вопрос 1: Что такое строка в Python?

1. Строка в Python — это линия, которая проходит через код.
2. Строка в Python — это простой текст, заключенный в кавычки.
3. Строка в Python — это значение с форматом целого числа.
4. Строка в Python — это значение с форматом вещественного числа.

Вопрос 2: Что нужно сделать, чтобы создать строку?

1. Нужно заключить текст в фигурные скобки.
2. Нужно заключить текст в круглые скобки.
3. Нужно заключить текст в кавычки.

Вопрос 3: Расставьте код так в правильной последовательности, чтобы получить переменную со значением в виде строки. А затем вывести значение переменной на экран.

1. message
2. =
3. "Привет Илон Маск!"
4. print
5. (message)

ГЛАВА ПЯТАЯ: БУЛЕВА ЛОГИКА

Булева логика в Python, как и в других языках программирования нужна нам для того, чтобы наш код мог сравнивать данные.

Например сравнивать пароль, который вводит пользователь, с паролем, который храниться в базе данных. Или сравнивать ранг игроков в видеоигре, чтобы подбирать соперников по уровню.

Таких примеров сравнений много, и здесь булева логика — крайне полезная штука.

Давайте узнаем из чего она состоит и какие примеры с ней можно создавать.

1. Операторы сравнения

Когда мы сравниваем числа друг с другом, мы обычно используем такие символы, как > (больше), < (меньше), = (равно) и так далее. Они работают и на Python. И вот как они выглядят там:

Больше >
Меньше <
Равно ==
Не равно !=
Больше или равно > =
Меньше или равно < =

Как Вы заметили, в Python вместо знака равенства = нам необходимо использовать двойной знак равенства ==.

Нам нужен двойной знак равенства для того, чтобы Python не

подумал, что мы пытаемся создать переменную.

В Python эти символы называются операторами сравнения. И они нужны нам для того, чтобы наша программа могла сравнивать различные типы данных, например: числа, строки, переменные и так далее.

2. Булевы значения: правда или ложь

У Булевой логики есть логические значения: True и False:

True — когда условие истинно. И False — когда условие ложно.

Давайте теперь сыграем.

Я буду сравнивать числа, а Вы будете отвечать в уме — True, если это истина, или False, если это ложь.

Начнем?

2 > 4
10 < 20
3 == 3
5 != 7

А теперь давайте сделаем то же самое в Python. Для этого мы создадим переменную — результат, и присвоим ей значение 2 > 4

Затем выведем результат на экран с помощью функции print. Вот, как это будет выглядеть:

```
результат = 2 > 4  
print(результат)
```

Впишите этот код в компилятор и запустите его.

Если значение переменной является истиной, программа вернет результат True. Если же значение окажется ложью, программа вернет False.

Ну как? Что возвращает программа?

А теперь подставьте остальные значения из списка, одно за

другим, запускайте код и наблюдайте за результатом, который возвращает программа.

Вот так и работает булева логика в Python.

Но что, если мы попробуем сравнить строки? Попробуйте сравнить яблоки с апельсинами, используя следующее значение в переменной:

```
результат = "яблоки" == "апельсины"  
print(результат)
```

Запишите этот код в компилятор и запустите его.

А теперь сравните яблоки с яблоками:

```
результат = "яблоки" == "яблоки"  
print(результат)
```

Видите результат?

Теперь Вы знаете, что в программировании булева логика работает не только с числовыми, но и с другими форматами данных.

3. True и False в переменных

Как Вы уже заметили, мы можем хранить результаты сравнений, которые возвращают True или False в переменных.

Давайте рассмотрим эту тему подробнее вот с таким примером:

```
результат = "яблоки" == "бананы"  
print(результат)
```

Давайте впишем этот код в компилятор и запустим его.

В этом коде мы создали переменную с именем “результат”, и присвоили ей логическое значение от сравнения двух строк «яблоки» и «бананы».

Затем мы спустились на одну строку вниз, и вывели значение переменной “результат” на экран, передав имя переменной в скобки

функции print. Которую мы перед этим создали.

Правда, не сложно?

А теперь измените код, указав, что яблоки и бананы не равны. Вы уже знаете как это делать :)

Что теперь возвращает программа?

4. Сравнение переменных в Python

Мы можем сравнивать не только числа и строки, но и целые переменные!

Посмотрите, как это можно сделать:

```
игра = "Dota"  
результат = игра == "FIFA"  
print(результат)
```

Но, прежде чем записать это пример в компилятор и запустить его, подумайте и скажите, какой результат он вернет? True или False?

Решили? А теперь давайте посмотрим правильно Вы решили, или нет:

Как видите, сначала мы создали переменную под названием игра и присвоили ей значение строки "Dota".

Затем мы спустились на одну строку ниже, и создали вторую переменную по имени результат.

После этого мы присвоили переменной результат логическое значение от сравнения нашей первой переменной — игра, со строковым значением FIFA.

Затем мы спустились еще на одну строку ниже, и вывели значение переменной результат на экран, с помощью функции print.

А так как значение нашей переменной игра равно строке Dota, а не строке FIFA, то программа вернула False.

Ну как? Совпало Ваше решение с ответом программы?

Давайте теперь закрепим пройденный пример. Возьмите наш код, и напишите на его основе свою версию программы для создания и сравнения двух переменных.

Меняйте что угодно по вашему желанию: имена переменных, их значения. И конечно операторы сравнения.

5. Программа для проверки пароля

Ну и в завершение пройденной главы, давайте отработаем очень упрощенный пример из реальной жизни: Программа, которая проверяет правильность введенного пароля:

```
пароль = "рыба-меч"  
приветствие = пароль == "рыба-меч"  
print(приветствие)
```

Введите этот код в компилятор и запустите его.

Давайте рассмотрим все по порядку.

В первой строчке мы создали переменную — пароль, и присвоили ей значение — рыба-меч.

Затем на второй строчке мы создали новую переменную по имени приветствие. И присвоили ей логическое значение, которое сравнивает значение переменной пароль и строку “рыба-меч”.

И, наконец, в третьей строчке мы вывели на экран результат логического сравнения из второй строчки.

Поскольку в качестве оператора сравнения мы использовали == (Равно), а значение переменной пароль действительно равно строке “рыба-меч”, то, переменная приветствие вернула True.

Таким образом, можно представить, что:

“рыба-меч” из первой строчки — это пароль, который хранится в базе паролей.

“рыба-меч” из второй строчки — это пароль, который вводит

пользователь, чтобы войти в личный кабинет.

приветствие = пароль == из второй строчки — это код, который сверяет пароль от пользователя с паролем из базы.

А `print(приветствие)` из третьей строчки — это всего лишь функция, которая выводит результат сверки на экран. `True`, если пароли совпадают, или `False`, если пароли не совпадают.

Теперь измените “рыба-меч” из первой или второй строчки на любое другое слово или словосочетание, и перезапустите программу. Программа должна вернуть `False`, так как пароли больше не совпадают.

Этот пример является упрощенным представлением того, как программисты используют булеву логику и операторы сравнения в Python для задач, связанных с авторизацией пользователей в приложениях.

6. Итоги пятой главы

В пятой главе Вы сделали следующее:

1. Узнали, что такое булева логика в Python;
2. Научились применять операторы сравнения;
3. Познакомились с логическими значениями `True` и `False`;
4. Научились использовать логические значения в переменных;
5. Узнали, как программисты используют булеву логику для авторизации пользователей.

Поздравляю! У Вас отличный прогресс! А теперь давайте применим булеву логику в более сложных примерах.

Это мы сделаем в следующей главе — Условные заявления в Python.

7. Тест пятой главы

Вопрос 1: Этот код хранит результаты сравнения двух строк, «яблоки» и «апельсины», в переменной по имени `результат`, а затем выводит значение переменной на экран. Но код перепутался. Расположите фрагменты кода в правильном порядке.

1. Результат
2. "апельсины"
3. (результат)
4. =
5. "яблоки"
6. print
7. !=

Вопрос 2: Этот код сравнивает две переменные. Какой результат вернет код, когда мы запустим программу?

```
машина = "Tesla"  
результат = машина == "Toyota"  
print (результат)
```

Вопрос 3: У булевой логики есть логические значения. Их всего два. Подставьте правильные определения для каждого из двух значений.

Когда условие оказывается правдой
Когда условие оказывается неправдой

True
False

ГЛАВА ШЕСТАЯ: УСЛОВНЫЕ ЗАЯВЛЕНИЯ

Условные заявления в Python, как и в других языках программирования позволяют нашей программе проверять определенные условия, и выполнять различные инструкции. В зависимости от того, сработало условие или нет.

Например можно создать такое условие и инструкцию: Если пароль верный, пустить пользователя на сайт. Если пароль не верный, выдать сообщение "Пароль не верный"

И так далее.

Условные заявления применяются в видеоиграх, на веб-сайтах, в мобильных приложениях и практически везде. И нам часто придется использовать их в своих программах.

А раз так, давайте научимся их применять!

1. Что такое условные операторы?

Мы проверяем условия, и даем себе инструкции каждый день:

Если сегодня солнечно, то надену очки.

Если в кафе есть WiFi, то спрошу у официанта пароль.

Как Вы заметили, оба примера сверху содержат условие, которое начинается с если (если сегодня холодно, если в кафе есть WiFi), и инструкции (надену очки, спрошу пароль).

Компьютерные программы тоже проверяют условия, чтобы следовать конкретным инструкциям.

Если условие срабатывает, программа возвращает True (помните True и False?), и выполняет одну инструкцию.

Если условие не срабатывает, программа возвращает False, и выполняет другую инструкцию.

Например, когда Вы хотите зайти на сайт, и вводите верный пароль, программа возвращает True, и выполняет инструкцию, которая говорит ей впустить вас.

Если Ваш пароль неверный, программа возвращает False, и следует другой инструкции, которая говорит ей не впускать вас.

Это и есть упрощенное представление того, как работают условные заявления в Python, и в других языках.

Программа проверяет условие и следует заявленной инструкции, в зависимости от того, сработало условие, или нет.

Так легче контролировать ход работы программы. Кстати, ход работы программы называется “Control flow”.

2. Условный оператор If

If — это условный оператор. Он переводится как “Если”.

Мы используем его, когда говорим программе, что нужно сделать, если условие сработало.

А теперь давайте посмотрим, как надо работать с этим условным оператором.

Вот Вам пример для иллюстрации.

Это небольшая программа, которая проверяет Ваш пароль и, если пароль верный, то программа приветствует Вас на сайте.

```
пароль = "секрет"  
if пароль == "секрет":  
....print("Добро пожаловать на сайт!")
```

Впишите этот код в компилятор и запустите его.

Давайте разберем его подробнее.

На первой строчке мы указали условие: пароль = “секрет”

На второй строчке мы прописали оператор if, а затем условие, которое нужно проверить. То есть проверить, что пароль действительно равен значению “секрет”. И поставили двоеточие после условия;

И, наконец, на третьей строчке, мы прописали инструкцию вывести сообщение на экран на случай, если условие окажется верным, и программа вернет True.

При этом сама инструкция выводит сообщение “Добро пожаловать на сайт!” на экран.

Кстати, Вы заметили двоеточие в конце второй строчки?

Оно нужно для того, чтобы программа поняла, где заканчивается условие, которое мы задали, и где начинается инструкция.

Двоеточие всегда ставится после условия, и перед инструкцией.

И, наконец, Вы наверное обратили внимание на то, что в этом коде, инструкция начинается после отступа.

Отступ перед инструкцией нужен для того, чтобы программа поняла, что эта инструкция относится именно к этому условию.

Попробуйте убрать отступ, и запустите программу снова. Вы увидите, что программа вернула ошибку. И ругается на то, что нет отступа.

Не волнуйтесь, в нашем компиляторе отступы ставятся автоматически, после условий с двоеточиями.

2.1. if и операторы сравнения

Как Вы уже поняли из нашего примера с паролем, условный оператор if прекрасно работает с операторами сравнения.

Вот Вам еще один пример:

```
день = "Пятница"  
if день == "Пятница":  
....print("Ура! Пятница!")
```

Впишите этот код в компилятор и запустите его.

А теперь поиграйте с этим кодом. Замените == на != и перезапустите его. Что теперь выдает программа?

PS: Правильно, ничего теперь не выдает.

Потому что у программы есть инструкция на случай, если условие выполнено.

А на случай, если условие не выполнено, инструкции нет. Вот программа и замолчала.

Не переживайте. Чуть позже, в этом курсе Вы научитесь решать эту проблему. А пока продолжим.

2.2. Условный оператор if и числа

С помощью условного оператора if можно сравнивать не только строки, но и числа. Смотрите, как это можно сделать:

```
очки = 100
```



```
if очки == 100:  
....print("Вы выиграли!")
```

Запишите эту программу в наш компилятор, и запустите ее.

Давайте теперь разберем, что здесь к чему.

Сначала мы создали переменную очки, и присвоили ей значение в виде целого числа 100.

Затем мы написали условное заявление, в котором, если значение очков равно сотне, программа должна вернуть сообщение “Вы выиграли!”

С if мы можем использовать не только оператор равно, но и другие операторы сравнения. Например, такие операторы, как больше или меньше.

Давайте дополним нашу программу условием, в котором есть оператор — меньше.

Вот, как мы можем это сделать:

```
очки = 50  
if очки < 100:  
....print("Эй! Вам нужно больше очков!")
```

Впишите этот код в консоль, и запустите его. Что теперь возвращает программа?

А теперь, давайте сделаем то же самое с условием, в котором есть оператор больше.

```
очки = 101  
if очки > 100:  
....print("Ух ты! У Вас новый рекорд!")
```

3. Условный оператор else

Мы узнали, что такое оператор if, и что он делает. Теперь давайте изучим оператор “else”.

Оператор else сообщает программе, что делать, если первое

условие не сработало. Вот, как это выглядит на примере:

```
очки = 99
if очки == 100:
....print("Вы выиграли!")
else:
....print("Вам нужно больше очков!")
```

Запишите это программу в наш компилятор, и запустите ее.

Как Вы уже поняли, теперь у нас есть два условия и две инструкции, для каждого условия.

Первое условие создано при помощи оператора if, а второе — при помощи оператора else.

Приведу для Вас еще один пример. Но, на этот раз я специально кое-что упустил в нем.

```
жизни = 1
if жизни == 0:
....print("Игра окончена")
:
....print("Сыграть еще раз?"
```

Ваша задача вписать этот код в консоль, выяснить, чего не хватает, исправить это, и запустить наш код.

Ну как, получается? :)

4. Условный оператор elif

Вы уже знаете об условных операторах if, else, и для чего они нужны.

Пришло время познакомиться с условным оператором elif.

elif происходит от комбинации операторов else и if. Перевести elif можно как “Иначе если”.

Для чего нам нужен оператор elif?

elif помогает нам добавлять в программу дополнительные

условия и соответствующие инструкции.

Вот, как это выглядит на примере:

```
время = 12
if время <12:
....print("Доброе утро!")
elif время < 17:
....print("Добрый день!")
```

Впишите этот код в наш компилятор, и запустите его.

Давайте рассмотрим его поподробнее.

Здесь мы создали два конкретных условия и две конкретные инструкции.

Одно условие и инструкция стоят после if. Второе условие и инструкция стоят после elif.

Если с этим примером все понятно, возьмите мой код за основу, и напишите свою программу, которая использует оба оператора if и elif.

Поиграйте, меняйте и условия, и инструкции.

Возможно, Вы спросите, если у нас уже есть оператор else, зачем нам еще и elif?

Хороший вопрос! Дело в том, что в отличие от else, условный оператор elif можно использовать столько раз подряд, сколько конкретных условий и инструкций мы хотим прописать.

Вот, как выглядит пример с двумя elif подряд:

```
время = 12
if время <12:
....print("Доброе утро!")
elif время < 17:
....print("Добрый день!")
elif время < 21:
....print("Добрый вечер!")
```

Впишите этот код в наш компилятор, и запустите его.

Если с этим все понятно, вот Вам еще немного самостоятельной работы.

```
время = 12
if время < 12:
....print("Доброе утро!")
....время < 17:
....print("Добрый день!")
....время < 21:
....print("Добрый вечер!")
```

В этом коде кое-чего не хватает. Дополните код, и запустите его.

Справились? Отлично! А теперь поговорим о порядке применения if, elif и else.

5. Порядок if, elif и else

Когда мы используем условные заявления, оператор if всегда идет первым. Так мы сообщаем программе конкретное условие и инструкцию, которую программа выполнит, если условие работает.

Затем идет elif. Так мы сообщаем программе другие конкретные условия и конкретные инструкции по каждому из условий.

И, наконец, условный оператор else. Он идет в самом конце. Так мы сообщаем программе инструкцию, которой она должна следовать, если ни одно из условий выше не сработало.

Вот, как выглядит такой порядок на примере:

```
время = 22
if время < 12:
....print("Доброе утро!")
elif время < 17:
....print("Добрый день!")
elif время < 21:
....print("Добрый вечер!")
else:
....print("Спокойной ночи!")
```

Впишите этот код в наш компилятор, и запустите его.

Итак, запомните: Сначала if, затем elif, и только потом else.

6. Итоги шестой главы

В шестой главе Вы сделали следующее:

1. Узнали, что такое условные заявления и как они работают в Python;
2. Узнали, что такое условные операторы;
3. Освоили условный оператор if;
4. Применили оператор if с операторами сравнения;
5. Применили оператор if с числами;
6. Освоили условный оператор else;
7. Освоили условный оператор elif;
8. Узнали о порядке применения if, elif и else.

Поздравляю Вас с отличным прогрессом! Настало время собрать все что мы изучили до сих пор, и применить это в новой теме: Циклы.

7. Тест шестой главы

Вопрос 1: Для чего нужны условные заявления?

1. Условные заявления нужны для создания переменной.
2. Условные заявления нужны для того, чтобы программа знала, какие инструкции и при каких условиях она должна выполнять.
3. Условные заявления нужны для создания строк.
4. Условные заявления нужны для того, чтобы ставить программе свои условия.

Вопрос 2: Для чего мы используем оператор if?

1. Мы используем оператор if, для того, чтобы сказать программе что делать, если условие сработало.
2. Мы используем оператор if, для того, чтобы программа смогла сохранить файл.
3. Мы используем оператор if, когда еще не знаем, каким будет условие.
4. Мы используем оператор if для создания строки.

Вопрос 3: Для чего мы используем оператор else?

1. Мы используем оператор else, для того, чтобы сказать программе, что делать, если первое условие не сработало.
2. Мы используем оператор else, когда не хотим использовать оператор if.
3. Мы используем оператор else, для того, чтобы программа сообщила нам свои условия.
4. Мы используем оператор else, когда не хотим использовать оператор elif.

Вопрос 4: Для чего нам нужен оператор elif?

1. Elif нам не помогает. Нам помогает только if и else.
2. Elif помогает нам создать переменную и присвоить ей значение.
3. Elif помогает нам добавлять строку в наше условие.
4. Elif помогает нам добавлять в программу дополнительные условия и соответствующие инструкции.

Вопрос 5: Расставьте операторы в правильном порядке их использования.

1. if
2. else
3. elif

Вопрос 6: Расставьте код в правильном порядке.

1. elif время < 17:
2. else:
3. if время < 12:
4.print("Добрый вечер!")
5. время = 12
6.print("Доброе утро!")
7.print("Доброй ночи!")
8.print("Добрый день!")
9. elif время < 21:

ГЛАВА СЕДЬМАЯ: ЦИКЛЫ

Циклы в Python, как и в других языках программирования помогают нам избежать повторения одних и тех же инструкций в

коде.

Они делают наш код более изящным и эффективным. Давайте узнаем как они выглядят и что эти циклы конкретно делают.

1. Что такое Циклы?

Очень часто, программы повторяют одну и ту же задачу, раз за разом. Например складывают числа пока не получится нужная сумма этих чисел.

Например, вот как выглядит программа которая повторяет счет до трех и выводит это на экран.

```
число = 1
print(число)
число = число + 1
print(число)
число = число + 1
print(число)
```

Впишите ее в компилятор, запустите и проверьте, что она выдаст.

Как видите, мы создали переменную с именем число, присвоили ей значение единица, а затем отображали ее на экране с помощью функции print.

Затем, чтобы увеличить число от одного до двух, мы снова взяли нашу переменную, присвоили ей текущее значение, добавили к нему 1, и снова отображали ее на экране с помощью функции print.

Затем мы повторили предыдущий шаг, чтобы получилось три.

Вроде работы немного. Но представьте, если бы нам пришлось считать до миллиона? Сколько строчек кода нам пришлось бы писать?

К счастью, для того, чтобы не повторять одни и те же инструкции в коде, программисты используют циклы.

Давайте возьмем реальный пример того, как они применяются.

Допустим, Вы играете в видеоигру-шутер со своим другом. Вы видите своего соперника, и начинаете стрелять из автоматического оружия.

Пока Вы удерживаете левую кнопку мыши, Ваше оружие продолжает стрелять.

Это происходит потому, что в программе используется цикл, который говорит следующее: «Стрелять, пока зажата кнопка».

Не будь этого цикла, Вам бы пришлось постоянно кликать мышью, чтобы продолжать стрельбу.

А еще циклы отвечают за движения в играх. Пока зажата клавиша “вперед”, игрок движется вперед.

Вам не приходится постоянно нажимать на клавишу. Вы просто держите ее зажатой, и цикл “двигаться вперед” непрерывно действует.

Итак, как Вы поняли, циклы — штука довольно полезная. Давайте теперь узнаем, как эти циклы применяются в Python.

2. Цикл `while`

Цикл `while` (или `while loop`) переводится как цикл “пока” и он работает пока определенное условие действует. (То есть возвращает `True`)

Вот, как выглядит этот цикл вместе с булевым значением `true`:

```
while true:
```

А теперь давайте добавим в этот цикл инструкцию:

```
число = 1
while True:
    ....print(число)
    ....число = число + 1
```

Запишите эту программу в наш компилятор, и запустите ее.

Программа не останавливается? :)

Как я уже говорил, while loop будет работать, пока определенное условие действует. (То есть возвращает True)

Сейчас мы узнаем как остановить цикл while, а пока жмите на красную кнопку стоп в нашем компиляторе.

3. Как остановить цикл while

Итак, как Вы поняли, пока цикл while возвращает True, он не остановится. Но как нам его остановить?

Давайте изменим наш предыдущий код так, чтобы он проиграл цикл один раз и остановил его. Вот как это выглядит:

```
сообщение = True
while сообщение == True:
....print("Один раз и хватит!")
....сообщение = False
```

Впишите эту программу в компилятор, и запустите ее.

Давайте разберем ее подробнее.

Сначала мы создали переменную “сообщение” и присвоили ей значение True.

Затем мы создали условие, используя цикл while. В этом условии сказано, что пока значение переменной “сообщение” равно True, программа должна выполнять инструкцию по выводу строки “Один раз и хватит!” на экран.

Затем мы создали значение False, и сохранили его в переменной “сообщение”.

Как Вы уже знаете, программа исполняет код построчно. Сверху вниз.

А значит, сначала программа видит переменную “message” и ее значение True.

Затем программа видит условие, которое запускает цикл по выводу строки “Один раз и хватит!” на экран, пока “message” равно

И наконец, спустившись на последнюю строчку, программа видит что переменная “message” больше не равна True и следовательно останавливает цикл.

4. Цикл while и операторы сравнения

Эта программа считает от одного до десяти. Благодаря циклу `while` и оператору сравнения, мы уложились всего в четыре строчки кода, чтобы написать такую программу.

А вот так бы выглядела наша программа без цикла while и операторов сравнения:

```
число = 1  
print(число)  
число = число + 1  
print(число)  
число = число + 1  
print(число)  
число = число + 1  
print(число)  
число = число + 1  
print(число)  
число = число + 1  
print(число)
```

```
число = число + 1  
print(число)  
число = число + 1  
print(число)
```

Правда, что циклы и операторы делают жизнь проще?

5. Цикл `while` и обратный отсчет

При помощи цикла `while` можно вести и обратный отсчет.

Обратный отсчет в программах — очень полезная штука.

Например, обратный отсчет используют в видеоиграх, чтобы считать сколько осталось времени, патронов, здоровья, защиты, противников. Ну Вы поняли.

А раз так, давайте напишем программу, которая ведет обратный отсчет от десяти до одного.

На этот раз, для того, чтобы объяснить Вам значение каждой строчки в коде, я написал комментарии над каждой строчкой.

Комментарии в Python пишутся после знака `#`. Благодаря комментариям программисты запоминают что, где и для чего они писали.

Не волнуйтесь, комментарии не мешают работе программы.

Python игнорирует любые символы, которые пишутся после знака `#`. Поэтому не забывайте его ставить, когда будете писать собственные комментарии.

Итак, вернемся к программе которая ведет обратный отсчет от десяти до одного:

Сначала приведу код без комментариев. Вот он:

```
число = 10  
while число >= 1:  
....print(число)  
....число = число — 1
```

Теперь закомментированный код:

```
# сохраняем 10 в переменной "число"
число = 10
# создаем цикл, который работает при условии, что больше или
равно 1
while число >= 1:
    # выводим значение числа на экран, пока работает цикл
    ....print(число)
    # отнимаем от числа по единице пока работает цикл
    ....число = число — 1
```

Впишите эту программу в компилятор, и запустите ее. Посмотрите, что она выводит.

А затем возьмите этот код за основу. И напишите свою собственную программу, которая использует цикл while и обратный отсчет.

Меняйте числа и операторы. Наблюдайте за тем, как меняется результат, который выводит программа.

6. Цикл for

Цикл for не использует условие, как это делает цикл while.

Вместо этого он повторяет фрагмент кода для каждого элемента из списка, пока список не закончится.

О том, что такое списки, для чего они нужны, и как с ними работать, мы поговорим в следующих главах.

А пока давайте посмотрим на пример программы, в котором есть цикл for:

```
for число in range(1, 11):
    ....print(число)
```

Эта программа считает от одного до десяти. Впишите ее в наш компилятор, запустите, и проверьте результат.

Заметьте, что если циклу while для работы нужно значение True, то циклу for требуется ключевые слова “in” и “range” как в нашем

примере выше.

А еще, обратите внимание на то, что последняя цифра в нашем списке не учитывается. Поэтому если мы хотим посчитать до 10, то должны указать 11.

Ну вот Вы и узнали как работает цикл `for` в Python. И если с этим все понятно, то напишите и запустите собственную программу, используя цикл `for` и ключевые слова `in` и `range`.

Если вам трудно, и у вас не получается, то просто посмотрите на пример моего кода. Меняйте числа и инструкции. Экспериментируйте!

7. Итоги седьмой главы

В седьмой главе Вы сделали следующее:

1. Освоили цикл `while`;
2. Научились останавливать цикл `while`;
3. Применили цикл `while` вместе с операторами сравнения;
4. Научились использовать цикл `while` для создания счета и обратного отсчета;
5. Освоили цикл `For`.

Вы прекрасно потрудились! Но, впереди нас еще ждет несколько очень интересных тем. И следующая из них — Списки. Давайте освоим ее!

8. Тест седьмой главы

Вопрос 1: Для чего программисты используют циклы?

1. Для того, чтобы зациклить код.
2. Для того, чтобы создать переменную и присвоить ей значение.
3. Для того, чтобы создать условное заявление.
4. Для того, чтобы не повторять одни и те же задачи в коде.

Вопрос 2: Как работает цикл `while`?

1. Цикл `while` работает хорошо.
2. Цикл `while` работает пока определенное условие действует. (То есть возвращает `True`)

3. Цикл while не работает по выходным.
4. Цикл while работает, пока определенное условие возвращает False.

Вопрос 3: Расставьте строки кода в таком порядке, чтобы он проиграл цикл while один раз и остановил его.

1.сообщение = False
2. while сообщение == True:
3.print("Все! Хватит")
4. сообщение = True

Вопрос 4: Эта программа считает от 1 до 10. Расставьте ее код в правильном порядке.

1. while число <=10:
2.print(число)
3. число = 1
4.число = число + 1

Вопрос 5: Эта программа ведет обратный отсчет от 10 до 1. Расставьте ее код в правильном порядке.

1. while число >= 1:
2.число = число — 1
3. число = 10
4.print(число)

Вопрос 6: Что из этого правда о цикле for?

1. Цикл for не использует условие, как это делает цикл while. Вместо этого он повторяет фрагмент кода для каждого элемента из списка, пока список не закончится.
2. Цикл for такой же как и цикл while.
3. Цикл for не используется в Python.
4. Цикл for использует условие, как это делает цикл while. Он не повторяет фрагмент кода для каждого элемента из списка, пока список не закончится.

ГЛАВА ВОСЬМАЯ: СПИСКИ

Списки позволяют нам хранить данные с разным форматом, и обращаться к этим данным, когда нам это необходимо.

В программировании, списки довольно распространенная вещь. Поэтому, давайте разберем их подробно. И узнаем, как эти списки выглядят, как их создавать и что именно они делают.

1. Введение в списки

Что такое списки, и как они выглядят в Python?

Начнем с того, что список в программировании называется "List". List означает "список" по-английски. Очевидно, правда?

Ну так вот, списки хранят данные (значения) с разным форматом, например строки, числа, и т.д.

Каждое такое значение в списке имеет свой порядковый номер. Этот номер называется индексом.

Индекс нужен для того, чтобы программа поняла, о каком именно значении в списке идет речь.

Давайте посмотрим, как выглядит список с различными значениями, на примере кода:

```
счет = ["один", "два", "3", "четыре", "пять"]
```

А теперь давайте запишем этот список в наш компилятор, и выведем его на экран с помощью функции print.

```
счет = ["один", "два", "3", "четыре", "пять"]  
print(счет)
```

Что мы можем узнать про списки в Python из этого примера?

Во-первых, в начале и конце списка должны стоять квадратные скобки "[]";

Во-вторых, значения в списке разделяются запятой;

В-третьих, значения в списке имеют разный формат. Тут мы видим и строки и числа;

2. Индекс списка

Давайте разберемся в том, что такое индекс списка в Python.

Как Вы уже слышали, у каждого значения в списке есть свой индекс. И программа использует его, чтобы понять о каком именно значении из списка идет речь.

Давайте посмотрим как работает присвоение индекса значениям списка.

0	1	2	3	4
<hr/>				
["один", "два", "3", "четыре", "пять"]				
<hr/>				
0	-4	-3	-2	-1

Мы взяли список из прошлого примера, и присвоили каждому из пяти значений списка свой индекс.

Обратите внимание на то, что у каждого значения есть положительный и отрицательный индекс.

Положительный индекс начинает отсчет с нуля, и идет слева направо.

Отрицательный индекс начинает отсчет с -1, и идет справа налево.

Отрицательный индекс всегда содержит отрицательное число. То есть число с минусом.

Обязательно помните о том, что положительный индекс списка в Python начинается с нуля. В программах отчет всегда начинается с нуля. Многие новички забывают об этом.

3. Как извлечь значение из списка

Иногда, создавая какую нибудь программу, нам нужно создать список данных (значений), чтобы затем при необходимости извлечь значение из списка, то есть данные, которые нам нужны.

Для этого мы просто сообщаем программе индекс нужного

значения, и она возвращает нам это значение из списка.

Точно так же, как в ресторане фаст-фуда, Вы даете работнику номер Вашего заказа, и работник отдает вам Ваш бургер.

Ничего сложного, правда?

А теперь, давайте сообщим программе индекс значения, и посмотрим, что она нам вернет.

Вот, как мы это сделаем:

```
счет = ["один", "два", "3", "четыре", "пять"]  
print(счет[0])
```

Запишите этот код в наш компилятор, и запустите его.

А теперь давайте разбираться.

В этом коде, на второй строчке мы передали индекс нужного нам значения в функцию print.

Для этого в скобках функции мы прописали имя списка, затем открыли квадратные скобки, указали индекс, закрыли квадратные скобки, и закрыли скобку функции.

Как Вы видите, программа вернула нам первое значение “один”.

Как Вы уже знаете, нумерация индекса, как и любой счет в компьютерной программе, начинается с нуля.

Значение “один” в нашем списке соответствует нулю. Вот вам снова иллюстрация для наглядности:

0	1	2	3	4
["один", "два", "3", "четыре", "пять"]				
0	-4	-3	-2	-1

Поэтому программа и вернула нам “один”.

А как нам вернуть значение “четыре”? Измените наш код так,

чтобы программа вернула нам это значение.

Справились с задачей? Отлично!

Теперь давайте посмотрим, как нам вернуть более одного значения из списка.

Для этого нам нужно перечислить индекс нужных значений через запятую. Вот, как это выглядит:

```
счет = ["один", "два", "3", "четыре", "пять"]  
print(счет[0], счет[2])
```

Этот код возвращает нам первое значение — строку “один”, и третье значение — целое число “3”.

Теперь Ваша очередь. Попробуйте извлечь все остальные значения из нашего списка.

4. Объединение значений списка

Как выполнить объединение значений списка? Помните, как мы объединяли строки знаком +?

Мы можем выполнить объединение значений списка таким же способом. Давайте это сделаем:

```
счет = ["один", "два", "3", "четыре", "пять"]  
print(счет[0] + ", " + счет[2])
```

Смотрите. Здесь мы склеили не только два значения списка. Мы еще и поместили дополнительную строку в виде знака запятой между этими значениями.

Давайте пойдем немного дальше, и добавим еще одну строку для того, чтобы у нашего списка появилось название:

```
счет = ["один", "два", "3", "четыре", "пять"]  
print("Счет: " + счет[0] + ", " + счет[2])
```

Запишите эту программу в наш компилятор и запустите ее. Посмотрите, как список преобразился.

А затем поэкспериментируйте с ним. Попробуйте склеить что-нибудь еще, и добавьте в список новые значения.

5. Функции списка

Давайте посмотрим, что такое функции списка в Python, и для чего они нужны?

Со списком можно работать по-разному. Можно не только извлекать из него значения, но и, например, считать количество значений в списке.

Для этого программисты используют различные функции списка.

Функций списка довольно много. Давайте познакомимся с некоторыми из них.

6. Функция len

Что такое Функция len, и для чего она нужна?

len() — это сокращенное название от слова length, что переводится с английского как “длина”.

И делает она именно то, чем она и названа, возвращает нам длину списка.

Это удобно, если нам нужно подсчитать количество значений в списке. То есть, узнать его длину.

Мы уже знаем, что длина нашего списка — 5 значений. Давайте это проверим при помощи функции len(). Вот, как это делается:

```
счет = ["один", "два", "3", "четыре", "пять"]  
print(len(счет))
```

Запишите эту программу в компилятор, и запустите ее. У Вас тоже получилось число 5?

А теперь все по порядку:

В этом коде, на второй строчке, мы поместили название функции len в скобки другой функции (print).

Затем открыли скобки функции len, и поместили туда имя списка.

Затем закрыли скобки функции len, и закрыли скобки функции print.

Вот мы и разобрались с тем, как работает функция len в Python.

7. Функция append

Давайте теперь узнаем о том, что такое функция append, и для чего она нужна.

append() переводится с английского как “добавлять в конец”.

И это именно то, что эта функция и делает: она добавляет значения в конец нашего списка.

Давайте теперь что-нибудь добавим в наш список:

```
счет = ["один", "два", "3", "четыре", "пять"]
счет.append("шесть")
print(счет)
```

В этом коде на второй строчке мы указали название нашего списка, затем поставили точку, и без пробела прописали имя функции append.

Затем открыли скобки функции, и вставили туда шестое значение — строку “шесть”.

После чего закрыли скобки. Спустились на строчку ниже, и прописали функцию print, указав в скобках функции имя нашего списка.

Готово! Теперь у нас целых шесть значений в списке! Вот, как работает функция append в Python.

Пожалуйста, запишите этот код в компилятор, и запустите его. А затем добавьте в наш список еще несколько значений, используя функцию append.

Сделали? Получилось? Отлично! Идем дальше.

8. Объединение списков в Python

А теперь мы научимся выполнять объединение списков в Python.

Помните, мы говорили о том, что в списках можно хранить любые типы данных?

Это действительно так. В списках мы можем хранить не только строки и числа, но и другие списки!

Давайте создадим два списка. Один с героями Marvel, другой с героями DC comics.

Затем создадим третий список, который будет объединять в себе оба списка. Вперед!

```
marvel = ["Iron Man", "Thor", "Captain America", "Hulk"]
DC = ["Superman", "Batman", "Wonder Woman", "Aquaman"]
герои = [marvel, DC]
print(герои)
```

Запишите этот код в компилятор, и запустите его. Посмотрите, что получилось.

А теперь давайте пойдем еще дальше. Извлечем героя "Batman" из третьего списка!

Вот, как это можно сделать:

```
marvel = ["Iron Man", "Thor", "Captain America", "Hulk"]
DC = ["Superman", "Batman", "Wonder Woman", "Aquaman"]
герои = [marvel, DC]
print(герои[1][1])
```

Давайте разберем этот пример в деталях:

Индекс списка `marvel` = 0, индекс списка `DC` = 1.

Индекс значения "Batman" в списке `DC` тоже равен 1. Потому что он второй в списке. Помните о том, что индекс начинается с нуля?

А, значит, чтобы извлечь нашего Бэтмена из списка `DC`, нам

потребовалось дважды указать [1]. Один для списка (DC) и еще один для нужного значения в списке (Batman).

Вот, как работает объединение списков в Python.

Запишите этот код в наш компилятор, и запустите его.

А затем поиграйте с ним. Добавляйте других героев. Создавайте другие списки, и объединяйте их. Программирование — это супер сила. Тренируйте ее!

9. Кортежи

Со списками мы разобрались. Пора узнать о том, что такое кортежи.

В английском языке кортеж называется Tuple (тюпл).

Кортежи очень похожи на списки. Но имеют несколько отличий:

1) Кортеж весит меньше, чем список

Кортеж будет весить меньше, чем список с тем же количеством данных.

Это особенно полезно в том случае, когда речь идет о программах с большим объемом данных, которые нужно упорядочено хранить.

Чем меньше весит приложение, тем быстрее (скорее всего) оно будет работать. Тем быстрее Вы его скачаете. Тем меньше интернет-трафика Вы потратите на его скачивание. И тем меньше памяти это приложение займет на Вашем компьютере или телефоне.

2) Программа обрабатывает кортеж быстрее, чем список

Опять же, когда речь идет о большом объеме упорядоченных данных.

Программа быстрее найдет нужное Вам значение в кортеже, чем в списке. Даже если они содержат одинаковое количество данных.

Пользователям Ваших будущих программ и видеоигр вряд ли понравится ждать, пока Ваша программа или видеоигра найдет нужную информацию, или отреагирует на их команду. Не так ли?

3) Кортеж является неизменным типом данных

Помните, как мы меняли список, используя функцию `append`? Таким образом мы меняли оригинал списка.

С кортежем такой фокус не пройдет. Если мы внесем изменение в кортеж, то мы лишь изменим его копию, но не оригинал.

Таким образом, если к созданному списку программа имеет доступ для чтения и изменения, то, к кортежу программа имеет доступ только для чтения.

Такое свойство кортежа не так уж и плохо. Ведь оно защищает кортеж от случайных изменений.

4) Кортеж отличается от списка синтаксисом

Вместо квадратных скобок `[]` как у списков, кортежи используют круглые скобки `()`

На этом, пожалуй, все из основных отличий.

Теперь, когда мы поняли чем отличаются кортежи от списков, давайте запишем какие-нибудь-данные в кортеж!

Для этого мы просто возьмем один из наших предыдущих списков, и превратим его в кортеж:

```
marvel = ("Iron Man", "Thor", "Captain America", "Hulk")  
print(marvel)
```

Пожалуйста, запишите эту программу в компилятор, и запустите ее.

Как Вы видите, внешне кортежи похожи на списки. За исключением того, что они быстрее, легче, неизменны, и имеют круглые скобки вместо квадратных.

В остальном все работает, как в случае со списком. Например,

мы можем легко извлекать значения из кортежей. Таким же способом, каким мы это делали со списками.

10. Как превратить кортеж в список?

Хоть мы и “превратили” список в кортеж всего лишь изменив квадратные скобки на круглые, программисты делают это по другому.

И список, и кортеж — оба являются типами данных. И для того, чтобы изменить тип данных кортеж, на тип данных список, программисты используют функцию списка.

Эта функция называется list.

Мы уже говорили о функциях списка выше, и применяли некоторые из них.

Давайте теперь применим и функцию list. Она и поможет нам превратить кортеж в список.

Как мы уже знаем, list переводится с английского как “список”. И это именно то, что и делает эта функция. Превращает наш кортеж в список.

А теперь за работу:

```
marvel = ("Iron Man", "Thor", "Captain America", "Hulk")
список = list(marvel)
print(список)
```

Сначала мы создали кортеж, и сохранили его в переменной “marvel”.

Затем мы превратили кортеж в список, поместив название “marvel” в круглые скобки функции list(). И сохранили новый список в переменной “список”.

Затем вывели содержание списка на экран при помощи функции print()

Запишите эту программу в компилятор, и запустите ее. Проверяйте результат.

Тот факт, что значения, которые она вывела, находятся в квадратных скобках, говорит о том, что это список.

Ну как? Правда, не сложно превращать кортежи в списки, используя Python?

11. Как превратить список в кортеж?

Итак, как нам теперь преобразовать список в кортеж? Спойлер: точно так же, как мы преобразовали кортеж в список, но на этот раз с помощью функции `tuple()`.

Вот, как мы это сделаем:

```
marvel = ["Iron Man", "Thor", "Captain America", "Hulk"]
кортеж = tuple(marvel)
print(кортеж)
```

Давайте рассмотрим этот пример поподробнее.

На этот раз мы создали список, и сохранили его в переменной “marvel”.

Затем мы превратили список в кортеж, поместив название “marvel” в круглые скобки функции `tuple()`. И сохранили новый кортеж в переменной “кортеж”.

Затем, мы вывели содержание кортежа на экран при помощи функции `print()`

Запишите эту программу в компилятор, и проверьте результат. Вы видите круглые скобки на экране? Это означает, что мы успешно превратили список в кортеж.

Как Вы поняли, превращать кортеж в список также легко, как и превращать список в кортеж.

А теперь создайте пару своих примеров с кодом, который превращает списки в кортежи, и наоборот. Потренируйтесь.

12. Итоги восьмой главы

В восьмой главе Вы сделали следующее:

1. Узнали о списках и индексах в Python;
2. Научились извлекать значения из списка;
3. Научились объединять списки и значения списков;
4. Узнали о функциях списков, таких как `len` и `append`;
5. Узнали о кортежах;
6. Научились преобразовывать кортеж в список, и наоборот.

Глава со списками в Python получилась довольно объемной. Но, вы все равно его прошли! Поздравляю Вас!

Обещаю, что следующая глава посвященная словарям в Python будет короче. Давайте приступим к ней!

13. Тест восьмой главы

Вопрос 1: Что делают списки в Python?

1. Списки помогают нам ничего не забыть, при покупках в магазине.
2. Списки хранят данные (значения) с разным форматом, например строки, числа и тд.
3. Списки хранят номера телефонов наших друзей.
4. Списки помогают нам сохранять значения переменных.

Вопрос 2: Зачем списку нужен индекс?

1. Индекс не нужен списку.
2. Индекс нужен для того чтобы программа поняла о каком значении в списке идет речь.
3. Списки не имеют индекс.
4. Индекс нужен для того чтобы превратить список в кортеж.

Вопрос 3: Что из этого, список?

1. ("Iron Man", "Thor", "Captain America", "Hulk")
2. ["Iron Man", "Thor", "Captain America", "Hulk"]
3. "Iron Man", "Thor", "Captain America", "Hulk"

Вопрос 4: Что из этого — правда о списках? (множественный

выбор)

1. В начале и конце списка должны стоять квадратные скобки [].
2. Значения в списке разделяются запятой.
3. В начале и конце списка должны стоять круглые скобки.
4. Значения в списке могут иметь разный формат.
5. Значения в списке должны иметь одинаковый формат.

ГЛАВА ДЕВЯТАЯ: СЛОВАРИ

1. Что такое словари?

Словари в Python очень похожи на списки и кортежи. Только они не используют индекс значений. Вместо этого они используют пары, состоящие из ключа и значения.

Для того, чтобы понять о чем идет речь, просто представьте себе обыкновенный языковой словарь. Допустим, мы взяли англо-русский словарь.

Открываем его, и видим английское слово “hello” (это ключ) и его значение на русском “привет” (это значение ключа).

Вот, как выглядит словарь в Python:

```
персонажи = {"Batman": "Первый игрок", "Superman": "Второй  
игрок"}  
print(персонажи)
```

Видите? Для того, чтобы создать словарь, нам всего лишь нужно:

Прописать пару: ключ-значение, между которыми стоит двоеточие;

Поместить пару в фигурные скобки;

Если пар несколько, разделить каждую пару запятой;

И сохранить полученный словарь в переменной.

Запишите эту программу в консоль, запустите ее, и посмотрите, что она выдаст.

А теперь поменяйте пары на что-нибудь свое. Можете увеличить словарь и добавить больше пар.

Экспериментируйте, играйте!

2. Формат словаря в Python

Поговорим про формат словаря в Python.

Как Вы заметили из прошлого примера, и ключ, и значение в словаре имеют формат строк.

Но, мы можем использовать разные форматы внутри одной пары. Например, ключом может быть строка, а значением число. Или наоборот.

Давайте создадим новый словарь, в которой ключом будет строка, а значением целое число:

```
числа = {1: "один", 2: "два"}  
print(числа)
```

Запишите эту программу в компилятор, и запустите ее. Посмотрите, какими получились пары словаря.

А теперь снова измените пары под себя. Используйте другие числа и строки.

Посмотрите, как меняется ответ программы.

3. Добавление пар в словарь

Поговорим про добавление пар в словарь.

После создания словаря мы всегда можем добавить в него новые пары.

Вот, как это делается:

```
числа = {1: "один", 2: "два", 3: "три"}  
числа [4] = "четыре"  
print(числа)
```

Запишите этот код в компилятор, и запустите его. Давайте посмотрим, что мы сделали:

Сначала мы создали словарь, и сохранили его в переменной “числа”;

Затем, строчкой ниже, мы прописали название переменной;

Затем, без пробела, открыли квадратные скобки, вписали туда ключ (“4”), закрыли квадратную скобку.

Поставили знак равно и указали значение ключа (“четыре”)

Готово! Затем мы просто вывели содержание словаря, передав имя переменной в которой мы его сохранили, в скобки функции print.

Попрактикуйтесь, добавьте пятую пару в свой словарь, и запустите код.

4. Удаление пар из словаря

А как выполнить удаление пар из словаря? Ведь иногда, работая над своей программой или видеоигрой, Вам придется не только добавлять пары в словарь, но и удалять их оттуда.

Для того, чтобы выполнить удаление пар из словаря в Python, программисты используют метод pop().

pop переводится с английского как “вырвать что-то”. Давайте “вырвем” третью пару из нашего словаря:

```
числа = {1: "один", 2: "два", 3: "три"}  
числа.pop(3)  
print(числа)
```

Запишите этот код в компилятор, и запустите его. Давайте снова посмотрим, что мы сделали:

Здесь на второй строчке, после названия переменной “числа”, мы ставим точку. Затем, прописываем метод pop().

В круглые скобки метода мы помещаем имя ключа из той пары,

которую хотим удалить.

Готово! Теперь выводим новое содержание словаря на экран, поместив имя переменной, в которой он хранится, в скобки метода `print()`

А теперь попробуйте сами! Удалите какую-нибудь ещё пару из Вашего словаря.

5. Как получить значение по ключу?

Иногда программистам необходимо получить значение ключа из словаря.

И для того, чтобы получить значение, все, что нам нужно сделать, это сообщить программе имя ключа.

Вот, как это делается:

```
числа = {1: "один", 2: "два", 3: "три"}  
print(числа [1])
```

Давайте запишем этот код в компилятор, и запустим его. А теперь рассмотрим его подробнее.

В приведенном выше примере мы просто поместили имя ключа в квадратные скобки. И передали его вместе с именем словаря в функцию `print()`.

А теперь, создайте свой собственный словарь и извлеките из него любые значения.

6. Как получить ключ по значению?

Мы можем получать не только значение по ключу из словаря. Но и ключ по значению из того же самого словаря.

Для этого программисты используют метод `keys()`, что по-английски означает “ключи”.

Давайте применим метод `keys()` и извлечем ключи из нашего словаря:

```
числа = {1: "один", 2: "два", 3: "три"}  
print(числа.keys())
```

Запишите этот пример в компилятор, и запустите его. Как видите, это не сложно:

Мы просто указали название переменной, в которой сохранен словарь с ключами.

Поставили точку, прописали метод `keys()`. И поместили все это в скобки функции `print`.

А теперь попробуйте создать свой словарь, и извлечь ключ сами.

7. Проверка ключа в словаре

Проверка ключа в словаре позволяет нам проверить, есть ли конкретный ключ в этом самом словаре.

Для этого нам нужно использовать ключевое слово `in`. Помните, мы уже использовали его с циклами?

Давайте попробуем проверить, есть ли ключ 3 в нашем словаре:

```
числа = {1: "один", 2: "два", 3: "три"}  
print(3 in числа)
```

Запишите этот пример в наш компилятор, и запустите его.

Если в нашем словаре есть 3, а оно там есть, то программа вернет `True`.

Получилось? Отлично. Теперь разберем пример.

На второй строчке мы прописали ключ 3, указали `in`, затем указали имя словаря. Получилось `3 in числа`. И поместили все это в скобки функции `print()`

Если с этим примером все понятно, попробуйте поискать ключ, которого там точно нет. Посмотрите что вернет программа.

Посмотрите, что возвращает программа.

8. Как проверить длину словаря?

В заключении этой главы поговорим о том, что такое проверка длины словаря, и как ее применять.

Как мы уже поняли, словари очень похожи на списки.

Мы даже можем применять к словарю те же функции, что мы уже применяли к спискам.

Давайте применим функцию `len()` для того, чтобы посчитать сколько пар в словаре:

```
числа = {1: "один", 2: "два", 3: "три"}  
print(len(числа))
```

Запишите этот пример в наш компилятор, и запустите его.

Как видите, со словарем функция `len()` работает так же, как и со списком:

Мы указываем имя функции, помещаем в ее скобки название переменной, в которой хранится словарь.

А затем помещаем все это в скобки функции `print`. Для того, чтобы вывести количество пар словаря на экран.

Ну вот. Теперь, когда Вы знаете, как выполняется проверка длины словаря, поэкспериментируйте. Увеличьте количество пар в словаре, и снова примените функцию `len()`.

9. Итоги девятой главы

В девятой главе Вы сделали следующее:

1. Научились создавать словари в Python;
2. Узнали о форматах данных, которые используются в словарях;
3. Научились добавлять и удалять пары из словаря;
4. Научились получать значение по ключу и ключ по значению;
5. Проверили, есть ли в словаре ключ;
6. Узнали, как проверять длину словаря.

Поздравляю! Вы прекрасно справились со словарями, раз Вы дошли до этого места :)

А значит и с функциями в Python в нашей следующей главе у Вас не будет никаких проблем. Приступим!

10. Тест девятой главы

Вопрос 1: Что из этого — правда о словарях? (множественный выбор)

1. Словари очень похожи на списки и кортежи. Только они не используют индекс значений. Вместо этого они используют пары состоящие из ключа и значения.

2. После создания словаря, мы больше не можем добавлять в него пары.

3. Мы можем использовать разные форматы внутри одной пары. Например ключом может быть строка а значением число. Или наоборот.

Вопрос 2: Чем словари похожи на списки? (множественный выбор)

1. У них одинаковые методы добавления и обновления значений.

2. И списки и словари заключают свои значения в фигурные скобки.

3. Значения и списков и словарей могут иметь разный формат.

Вопрос 3: Какой или какие из этих способов действительно создают словарь? (множественный выбор)

1. `batman = {"возраст": 36, "имя": "Bruce Wayne"}`

2. `superman = {"имя": "Kal-L"}`

3. `cities = ["New York", "Tokyo"]`

4. `years = (2030, 2040)`

Вопрос 4: Соберите фрагменты кода в таком порядке чтобы программа вывела сообщение “Batman”

1. `dict["герой"] = "Batman"`

2. `....print(сообщение)`

3. `dict = {}`

4. `....сообщение = dict.pop("герой")`

5. if "герой" in dict.keys():

ГЛАВА ДЕСЯТАЯ: ФУНКЦИИ

Функции в Python Вам уже немного знакомы. Вы использовали некоторые из них в предыдущих главах.

Давайте теперь познакомимся с ними поближе. Узнаем подробнее о том, для чего они применяются, и как нам с ними работать.

1. Введение в функции

Функция нашего желудка — переваривать пищу. Функция наших глаз — видеть все, что нас окружает. А функция сторожевой собаки — охранять территорию.

В компьютерном коде тоже могут быть функции. Много функций.

Например, есть функция, которая выводит информацию на экран. Называется эта функция `print()`

Вы ее уже хорошо знаете. Не так ли?

А еще Вы знаете функцию `list()`, которая превращает кортеж в список. И функцию `tuple()` которая, наоборот, превращает список в кортеж()

Функции в Python бывают встроенными и пользовательскими.

Встроенные (build in), такие как `print()`, `list()`, `tuple()` уже есть в Python. И мы просто можем ими пользоваться.

Пользовательские (custom) функции — это те функции, которые мы можем создать сами. Если нам не хватает встроенных функций.

Для того, чтобы создать такую функцию, нам необходимо использовать ключевое слово `def`.

`def` — сокращение от `define`, что в переводе с английского означает “определить”.

А значит, когда мы создаем функцию, мы тем самым

“определяем” ее обязанности.

Давайте посмотрим как это выглядит на примере кода:

```
def поздоровайся():  
....print("Привет!")  
  
поздоровайся()
```

Запишите эту программу в наш компилятор, и запустите ее.

Получилось вывести сообщение на экран? Отлично! А теперь, давайте разбираться.

Сначала мы прописали ключевое слово `def`.

Затем прописали имя функции “поздоровайся”, поставили круглые скобки и точку с запятой.

Далее мы спустились на строчку ниже и сделали 3 отступа*

На этой новой строчке мы сказали нашей функции, что она должна делать. То есть выводить строку “Привет!”.

Для этого мы применили встроенную функцию `print()` и поместили строку “Привет!” в ее круглые скобки.

После чего спустились на 3 строчки ниже**.

И прописали имя нашей созданной функции, вместе с круглыми скобками. Это называется “вызвать функцию”

* После точки с запятой программа сама сделает 3 отступа, когда мы начнем новую строчку.

** Спускаться на 3 строчки ниже после определения функции — обязательно по правилам грамотного написания кода.

Если с этим все понятно, давайте перейдем к параметрам и аргументам функций.

2. Параметры и аргументы функций

Настало время поговорить про параметры и аргументы функций.

Как Вы заметили, сразу после имени функции и ее вызова идут круглые скобки. Их необязательно оставлять пустыми.

В скобки, стоящие сразу после имени функции, можно поместить параметр функции. А в скобки, стоящие после вызова функции, можно поместить аргумент функции.

Давайте посмотрим на пример ниже. Чтобы понять, где параметры, а где аргументы функций в Python:

```
def поздоровайся(имя):  
    ....print("Привет, " + имя + "!")  
  
поздоровайся("Илон Маск")
```

Слово “имя”, стоящее в скобках после поздоровайся — это параметр. А строка “Илон Маск”, стоящая в скобках после вызова функции — это аргумент.

Вы спросите, в чем разница между параметром и аргументом? И зачем они нам оба нужны?

Отличный вопрос! Ну так вот:

Параметр — это место, приготовленное для будущего значения. А аргумент — это и есть то значение, которое пришло на место, которое приготовил параметр.

Другими словами, параметр — это просто заполнитель. А аргумент — это то, что помещается в заполнитель.

Если с первого раза это трудно понять, не переживайте! Просто перечитайте несколько раз.

А теперь запишите этот код в наш компилятор, и запустите его.

```
def поздоровайся(имя):  
    ....print("Привет, " + имя + "!")
```

поздоровайся("Илон Маск")

Давайте разберем его подробно:

Мы поместили параметр под названием имя в круглые скобки функции.

Строчкой ниже мы склеили строки "Привет, " и "!" с именем параметра.

Теперь мы можем вызвать функцию, и передать ей аргумент с любым значением. В нашем случае это строка "Илон Маск".

Если с этим все понятно, потренируйтесь, замените "Илон Маск" на Ваше имя. И перезапустите код. Так Вы измените аргумент, подставив в него новое значение.

3. Как вернуть значение из функции

Возврат значений из функций — это обычная задача для программиста.

Давайте научимся это делать.

Для того, чтобы функция поняла, что ей нужно возвращать значение, нам необходимо использовать ключевое слово `return`, а затем прописать само значение.

`return` в переводе с английского означает "возвращать"

Для того, чтобы Вам было понятно как все это выглядит, я создал функцию под названием "терминатор". И ее обязанность — возвращать нам сообщение "Я еще вернусь..."

```
def терминатор(слова):  
....return "{}".format(слова)
```

```
сообщение = терминатор("Я еще вернусь...")  
print(сообщение)
```

Запишите этот пример в наш компилятор, и запустите его. Давайте внимательно его разберем:

Сначала мы прописали ключевое слово `def`. И назвали нашу функцию “терминатор”.

Затем мы открыли круглые скобки, и передали в них параметр под названием “слова”.

Строкой ниже, после трех отступов, мы прописали ключевое слово `return`.

А после `return` мы создали форматированную строку (помните эту тему?)

Так у нас получилась функция, которая возвращает значение при помощи форматированной строки.

Затем мы спустились на четвертую строчку, и “вызывали” функцию терминатор, прописав ее.

Затем, в качестве аргумента, мы передали ей строку “Я еще вернусь...”

И после этого мы сохранили вызов функции со значением “Я еще вернусь...” в переменной “сообщение”.

Наконец, мы спустились на строчку ниже, и вывели сообщение на экран.

Поработайте с этим кодом. Меняйте название функции, аргументов и значений. Перезапускайте код, и смотрите, как изменяется ответ программы.

4. Вложенные функции

Мы научились создавать пользовательские функции. Давайте теперь научимся создавать вложенные функции, и вложим одну функцию в другую.

Вот, как мы это сделаем:

```
def поздоровайся(имя):  
    ....print("Привет, " + имя + "!!")
```

```
def сообщение(имя, вопрос):
```

```
....поздоровайся(имя)  
....print("Скажи, " + вопрос)
```

```
сообщение("Илон", "когда летим на Марс?")
```

Перед тем, как мы рассмотрим этот пример во всех подробностях, пожалуйста, запишите его в наш компилятор, и запустите. Прочитайте ответ программы и подумайте, как мы его получили?

А теперь давайте рассмотрим наш код поподробнее.

На строчке 1 мы создали функцию под названием “поздоровайся”, и передали ей “имя” в параметр.

На строчке 2 мы вывели строки “Привет, ” и “!”, и соединили их с параметром функции.

В строчке 4 мы создали еще одну функцию под названием “сообщение”. Затем передали ей параметр первой функции (имя) и новый параметр (вопрос).

На строчке 5 мы вложили первую функцию во вторую, объявив имя первой функции с ее параметром.

На строчке 6 мы вывели строку “Скажи, ” и соединили ее со вторым параметром функции.

И, наконец, на строчке 8 мы вызвали вторую функцию, передав в ее аргумент значения для параметров первой и второй функции.

Если Вы не сразу все поняли, не переживайте. Понимание приходит с повторением и практикой.

Перечитывайте приведенный расклад столько, сколько Вам необходимо.

Меняйте код под себя, перезапускайте программу, и наблюдайте за тем, как меняется ее ответ.

5. Итоги десятой главы

В десятой главе Вы сделали следующее:

1. Узнали о функциях в Python;
2. Изучили параметры и аргументы функций;
3. Научились возвращать значения из функций;
4. Освоили вложенные функции.

Поздравляю Вас с завершением очередной главы! Мы движемся к завершению всей книги. Осталась пара тем и финальный проект. И следующая глава посвящена Классам и объектам. Приступим!

6. Тест десятой главы

Вопрос 1: Что из этого — правда о функциях? (Множественный выбор)

1. Функции бывают встроенные и пользовательские.
2. Для того чтобы создать функцию, нам необходимо использовать ключевое слово `def`.
3. В функцию нужно обязательно передавать параметр.
4. Название функции нужно указывать в кавычках.

Вопрос 2: Что из этого — правда о параметрах функций? (Множественный выбор)

1. Параметры в функциях могут иметь только численное значение.
2. Параметры в функциях указывать не обязательно.
3. В каждой функции может быть не более двух параметров.
4. Параметры могут иметь любое значение.

Вопрос 3: Соберите функции в таком порядке, чтобы программа вывела сообщение “Привет, Илон! Скажи, когда летим на Марс?”

1. `сообщение("Илон", "когда летим на Марс?")`
2. `def поздоровайся(имя):`
3. `....print("Скажи, " + вопрос)`
4. `def сообщение(имя, вопрос):`
5. `....поздоровайся(имя)`
6. `....print("Привет, " + имя + "!")`

Вопрос 4: В чем разница между параметром и аргументом?

1. Разница между параметром и аргументом не критична. И они взаимно-заменяемы.

2. Аргумент это место приготовленное для будущего значения. А параметр это и есть то значение, которое пришло на место, которое приготовил аргумент.

3. Параметр это место приготовленное для будущего значения. А аргумент это и есть то значение, которое пришло на место, которое приготовил параметр.

ГЛАВА ОДИННАДЦАТАЯ: КЛАССЫ И ОБЪЕКТЫ

Классы и объекты — одна из самых важных глав этого курса. И в ней Вы научитесь создавать классы. Узнаете о свойствах и функциях класса. Создадите объекты класса и научитесь ими управлять. Приступим!

1. Введение в классы и объекты

Python — это объектно-ориентированный язык программирования.

Это означает что, все, что мы до сих пор создавали с вами, является объектами.

А что такое классы?

Класс — это шаблон, по которому создаются все объекты. А объекты это — экземпляры класса.

Давайте посмотрим все на примере:

Баскетболист Стефен Карри — это объект (или экземпляр) класса “Баскетболист”.

Класс “Баскетболист” создает свои объекты по шаблону, в котором указаны свойства и функции этого класса. Эти свойства и функции класс передает своим объектам по наследству:

Свойства класса “Баскетболист”

Руки = 2;

Ноги = 2;

Рост > 1.9 м ;

И тд.

Функции класса “Баскетболист”

Передвигаться по баскетбольной площадке;

Бросать мяч в корзину;

Отбирать мяч у соперника;

Принимать мяч;

Пасовать мяч;

И тд.

Если с этим пока все понятно, давайте узнаем, как создать класс.

Да, и еще кое-что: по ходу дальнейших объяснений мы будем рассматривать примеры кода, которые нужно писать вместе со мной, для того, чтобы создать полноценный класс и объект.

Просто смотрите объяснения и пишите код вместе со мной, не запуская его. Как только настанет время запускать наш код, я об этом скажу.

Итак, поехали!

2. Как создать класс в Python?

Допустим, мы делаем игру. И нам необходимо создать соперников (rival по-английски).

У соперников функции и свойства одинаковые. А, значит, мы можем объединить их под одним классом “rival”.

Для того, чтобы создать класс, необходимо использовать ключевое слово class. Так Python поймет, что мы создаем класс.

Итак, создаем класс под названием Rival:

```
class Rival:
```

Пока запишите этот фрагмент в наш компилятор. Сейчас мы научимся создавать свойства класса.

3. Свойства класса

Пусть у соперника будет 3 жизни. Давайте для этого, прямо под классом, создадим переменную жизнь (life) со значением 3. Это и будет свойство класса Rival.

```
class Rival:  
....life = 3
```

Теперь нам необходимо создать функцию класса Rival. Давайте сделаем это.

4. Функции класса в Python

Одна из функций нашего соперника — принимать нашу атаку. Для этого создаем функцию класса под названием атака (attack).

Мы уже создавали функции, применяя ключевое слово def. Здесь поступаем так же. Пишем def, затем имя функции, открываем скобки и передаем аргумент (self)

Вот, как это выглядит в нашем примере:

```
class Rival:  
....life = 3  
  
....def attack(self):
```

“А что за self? Мы такое не проходили” Скажите Вы, и правильно сделаете.

self — это обязательное ключевое слово в Python, которое записывается в аргумент функции класса.

Позже, когда мы создадим объект класса, мы свяжем этот объект со своим классом именно через self. Вы это увидите на примере, и сразу все поймете.

А сейчас вернемся к функции.

Итак, мы создали функцию атаки (attack). Теперь скажем программе, что эта функция делает, а делает она вот что:

Каждый раз, когда она срабатывает, наш соперник будет кричать

“Ouch!”

Для этого в теле функции мы прописываем строку “Ouch!”, и передаем ее в аргумент функции print()

Вот, как это выглядит в нашем примере:

```
class Rival:
....life = 3

....def attack(self):
.....print("Ouch!")
```

Обратите внимание на отступы, которые мы делаем в коде:

Свойство класса — life и функция класса — attack помещаются в тело класса Rival с использованием четырех отступов.

При этом у функции тоже есть свое тело, и мы помещаем в него print(“Ouch!”), делая еще четыре отступа.

Так Python понимает к какому классу относятся свойства и функции. А еще так наш код выглядит более аккуратным и понятным.

А теперь вернемся к функции класса.

Каждая наша атака должна не только заставлять соперника кричать “Ouch!”, но и отнимать у него по одной жизни.

Для этого из значения переменной life нужно вычитать по одной единице.

Но, для того чтобы получить доступ к этой переменной класса, нам нужно прописать self и поставить точку. Так Python поймет, что внутри этого класса мы хотим отнять одну единицу в переменной life.

И вот, как это будет выглядеть в нашем примере:

```
class Rival:
....life = 3
```

```
....def attack(self):  
.....print("Ouch!")  
.....self.life -=1
```

Теперь функция атаки класса (attack) не только заставляет соперника кричать “Ouch!”, но и отнимает у него одну единицу жизни, каждый раз, когда мы его атакуем.

Теперь предлагаю создать еще одну функцию класса, которая будет считать количество жизней соперника после каждой нашей атаки.

Для этого спускаемся на две строчки ниже, и делаем то же, что и при создании предыдущей функции. А именно: пишем def, затем название этой функции (назовем новую функцию “checkLife”), и передаем ключевое слово self в ее аргумент.

Вот, что у нас получится на выходе:

```
class Rival:  
....life = 3  
  
....def attack(self):  
.....print("Ouch!")  
.....self.life -=1  
  
....def checkLife(self):
```

Итак, функцию мы прописали. Теперь давайте скажем программе, что делает эта функция.

Для этого мы воспользуемся условными заявлениям, о которых Вы уже знаете из пройденного материала.

Условные заявления для нашего класса:

Если количество жизней соперника будет меньше или равно нулю, функция выведет сообщение от имени соперника “You won!” (Вы победили!).

А если у соперника еще остались жизни, функция просто скажет, сколько жизней осталось.

Итак, сначала первое условие, в котором мы победили. Вот, как мы это условие пропишем:

```
class Rival:
....life = 3

....def attack(self):
.....print("Ouch!")
.....self.life -=1

....def checkLife(self):
.....if self.life <=0:
.....print("You won!")
```

Наш код становится немного сложнее, поэтому давайте подробно рассмотрим то, как мы создали это условие:

Под именем функции “checkLife”, после четырех отступов, мы прописали условный оператор if, поставили точку, и прописали ключевое слово self, чтобы программа снова поняла, что мы обращаемся к переменной life, класса Rival.

Затем мы прописали оператор сравнения меньше или равно (<=), поставили ноль и двоеточие.

Теперь нам надо создать инструкцию, которую программа запустит, если прописанное нами условие сработает.

Для этого мы спустились ниже, снова четыре отступа. И приказали программе вывести сообщение “You won!”. Для этого мы передали строку “You won!” в аргумент функции print().

Готово! Теперь, если количество жизней соперника меньше или равно нулю, функция выведет сообщение “You won!” от имени соперника.

Если с этим все понятно, давайте создадим второе условие, на тот случай если у нашего соперника еще остались жизни. И зададим соответствующую инструкцию на этот случай.

Второе условие для нашего класса:

```
class Rival:
```

```
....life = 3

....def attack(self):
.....print("Ouch!")
.....self.life -=1

....def checkLife(self):
.....if self.life <=0:
.....print("You won!")
.....else:
.....print(self.life)
```

Давайте снова рассмотрим подробно, что мы только что сделали:

Мы спустились еще на одну строку и присвоили условный оператор `else`.

Затем в следующей строке нужно было написать инструкцию для отображения оставшегося количества жизней.

Для этого мы назначили функцию `print`, и передали в ее аргумент ключевое слово `self` и имя переменной (`life`). Опять же, нам нужно `self`, чтобы программа знала, что мы имеем в виду переменную класса.

Ну вот! Мы создали простой класс `Rival`, где у нашего соперника три жизни. И где в ответ на каждую нашу атаку он кричит “Ouch!”, теряя по одной жизни.

А еще наш класс умеет считать и выводить количество оставшихся жизней после каждой атаки. И в случае, если жизней не осталось, торжественно выводит сообщение “You won!”

У нас получился довольно хороший класс. Давайте же теперь создадим объекты класса!

5. Как создать объект класса в Python

Что нужно знать, перед тем как создать объект класса в Python?

Во-первых, создание объектов класса очень похоже на создание переменной. Вы это сейчас увидите.

Во-вторых, создавая объект класса, нам необходимо дать нашему объекту название.

И в-третьих, поскольку класс у нас называется “соперник (Rival)”, то название объекта этого класса будет носить имя какого-нибудь злодея.

Поэтому, давайте назовем наш объект в честь Таноса (thanos) из Marvel. Достойный экземпляр соперника, не так ли?

Итак, приступим:

```
class Rival:
....life = 3

....def attack(self):
.....print("Ouch!")
.....self.life -=1

....def checkLife(self):
.....if self.life <=0:
.....print("You won!")
.....else:
.....print(self.life)

thanos = Rival()
```

Как Вы видите, мы начали с имени объекта.

После чего поставили знак равно, и указали название класса, к которому относится наш объект.

Далее мы просто закрыли имя класса круглыми скобками. Ничего сложного.

Теперь давайте присвоим объекту функцию класса, которая отвечает за подсчет жизни (checkLife).

Это позволит нашему объекту thanos проверить количество жизней перед схваткой.

Вот, как это выглядит в нашем примере:


```
class Rival:
....life = 3

....def attack(self):
.....print("Ouch!")
.....self.life -=1

....def checkLife(self):
.....if self.life <=0:
.....print("You won!")
.....else:
.....print(self.life)

thanos = Rival()

thanos.checkLife()
```

Давайте посмотрим, что мы сделали:

Мы спустились на две строки ниже, и прописали имя объекта.

После чего поставили точку, и прописали имя нужной нам функции (checkLife), которая уже имеется у нашего класса. Вы прописали ее на восьмой строчке.

Наконец-то настало время запускать наш код!

Запустите код, и посмотрите, сколько жизней насчитал у себя thanos перед схваткой.

Если Вы все сделали правильно, программа выдаст Вам цифру 3. Потому что мы задали Таносу три жизни перед схваткой.

А теперь в атаку! Давайте атакуем Таноса, и посмотрим, что он нам ответит.

Для этого мы спустимся на строчку ниже. Снова пропишем имя объекта thanos, поставим точку, и пропишем имя другой, знакомой нам функции, которую мы создали внутри класса. Это функция attack()

Вот, что у нас получится:

```
class Rival:
....life = 3
```

```
....def attack(self):
.....print("Ouch!")
.....self.life -=1

....def checkLife(self):
.....if self.life <=0:
.....print("You won!")
.....else:
.....print(self.life)
```

```
thanos = Rival()
```

```
thanos.checkLife()
thanos.attack()
```

Впишите `thanos.attack()` и запустите код. На это раз Таносу есть, что вам сказать!

Программа выдаст цифру 3, а под ней сообщение “Ouch!”

Откуда взялось сообщение “Ouch!”?

Правильно, от функции `print(“Ouch!”)`, которую мы ранее прописали внутри функции атаки — `def attack(self)`. Помните, да? Хорошо!

А сколько у Таноса осталось жизней после нашей первой атаки? Давайте узнаем это.

Для этого мы добавим в конец нашего кода еще одну функцию `checkLife()`. И вот, что у нас получится:

```
class Rival:
....life = 3

....def attack(self):
.....print("Ouch!")
.....self.life -=1

....def checkLife(self):
.....if self.life <=0:
.....print("You won!")
```

```
.....else:  
.....print(self.life)
```

```
thanos = Rival()
```

```
thanos.checkLife()  
thanos.attack()  
thanos.checkLife()
```

Впишите этот код в компилятор, и запускайте его. Если Вы все сделали правильно, программа выдаст цифру 3, затем сообщение “Ouch!”, а затем цифру 2.

А это значит, что у Таноса осталось две жизни! Мы побеждаем! Предлагаю атаковать нашего соперника еще два раза и посмотреть, кто победит.

Вот секретный код победы над злодеем:

```
class Rival:  
....life = 3  
  
....def attack(self):  
.....print("Ouch!")  
.....self.life -=1  
  
....def checkLife(self):  
.....if self.life <=0:  
.....print("You won!")  
.....else:  
.....print(self.life)
```

```
thanos = Rival()
```

```
thanos.checkLife()  
thanos.attack()  
thanos.attack()  
thanos.attack()  
thanos.checkLife()
```

Наконец, пришло время нанести решающий удар. Допишите этот код у себя, и запустите его.

PS: Писать функцию три раза подряд не самая лучшая практика. Но, я не хочу грузить Вас еще и циклами. В этой главе фокус только на классы и объекты.

Итак, если Вы все сделали правильно, программа выдаст Вам вот такое сообщение:

```
3
Ouch!
Ouch!
Ouch!
You won!
```

Ну как? Получилось?

Отличная работа! Вы только что научились создавать классы и объекты в Python! И даже победили Таноса по пути.

Но, тема классов и объектов будет неполной, если мы не узнаем, как управлять разными объектами, принадлежащими одному классу.

Давайте узнаем это.

6. Управление объектами класса

Теперь мы поняли что объект — это, по сути, копия своего класса.

Но, даже если несколько объектов и могут относиться к одному классу, между собой эти объекты все равно независимы.

Таким образом, если Вы отнимите жизнь у одного соперника (объекта своего класса), то это никак не повлияет на другого соперника (объекта того же класса).

На примере реальной жизни это выглядит так:

Вы играете в видеоигру, например, в Dota. Перед Вами два соперника: Slark и Axe, которые хотят вступить с Вами в ближний бой.

И Slark, и Axe — это объекты одного класса (класс соперников), у них даже есть общие функции, унаследованные от своего класса,

например, ближний бой.

Но при этом они остаются полностью независимыми друг от друга объектами.

Например, у каждого из них разные атрибуты: у Slark — ловкость, а у Ахе — сила.

А еще у каждого из них отдельная шкала здоровья. Если Вы нанесете урон Сларку, то его шкала здоровья изменится, а шкала здоровья Ахе останется без изменений.

Восстанавливаться их шкалы тоже могут с разной скоростью. И так далее.

Думаю, теперь Вы поняли, что программисты должны уметь управлять объектами класса так, чтобы эти объекты могли оставаться независимыми друг от друга.

А раз так, давайте создадим второй объект для нашего класса Rival, и сделаем его независимым от первого объекта.

Создаем второй объект класса

Первым объектом у нас был злодей thanos. Давайте добавим второй объект-злодея, пусть это будет Магнето (magneto).

Затем атакуем Таноса, и посмотрим как это отразится на Магнето.

Вот, как это будет выглядеть на примере:

```
class Rival:
....life = 3

....def attack(self):
.....print("Ouch!")
.....self.life -=1

....def checkLife(self):
.....if self.life <=0:
.....print("You won!")
.....else:
```

```
.....print(self.life)
```

```
thanos = Rival()  
magneto = Rival()
```

```
thanos.checkLife()  
magneto.checkLife()
```

Как Вы видите из примера, мы добавили второй объект, назвав его `magneto`, и присвоили ему класс `"Rival"`.

Затем, как и в случае с объектом `thanos`, мы присвоили ему функцию, которая проверяет жизнь: `magneto.checkLife()`

Таким образом, когда программа запускается, оба объекта проверяют и сообщают количество своих жизней.

Впишите этот код в компилятор, и запустите его. Если Вы все сделали правильно, программа дважды выведет цифру 3. Потому что и у Таноса, и у Магнето по три жизни.

Затем давайте атакуем Таноса три раза, после чего оба объекта снова проверят, и сообщат нам количество оставшихся жизней.

Для этого, все что нам осталось сделать, — это трижды добавить `thanos.attack()` в наш код.

Вот, как это выглядит на примере:

```
class Rival:  
    ....life = 3  
  
    ....def attack(self):  
        .....print("Ouch!")  
        .....self.life -=1  
  
    ....def checkLife(self):  
        .....if self.life <=0:  
            .....print("You won!")  
        .....else:  
            .....print(self.life)  
  
thanos = Rival()
```

```
magneto = Rival()
thanos.attack()
thanos.attack()
thanos.attack()
thanos.checkLife()
magneto.checkLife()
```

Впишите этот код в наш компилятор, и запустите его. Если Вы все сделали правильно, программа вернет Вам вот такое сообщение:

```
Ouch!
Ouch!
Ouch!
You won!
3
```

Как видите, Танос теперь поражен, но у Магнето все еще три жизни.

А это значит, что функции класса, которые мы применили к одному объекту, совершенно не влияют на другие объекты того же класса.

А теперь давайте отпразднуем пройденную тему небольшой самостоятельной работой.

Допишите код таким образом, чтобы победить и Магнето!

7. Итоги одиннадцатой главы

В одиннадцатой главе Вы сделали следующее:

1. Освоили классы и объекты в Python;
2. Узнали о свойствах и функциях класса;
3. Создали свои классы и объекты в Python;
4. Научились управлять разными объектами одного класса.

Вы проделали огромную работу, с чем я Вас и поздравляю! А теперь жду Вас в завершающей перед финальным проектом главе — Модули и пакеты.

8. Тест одиннадцатой главы

Вопрос 1: Что такое класс?

1. Класс — это шаблон, по которому создается объект.
2. Класс — это пример объекта.
3. Класс — это ключевое слово для создания переменной.
4. Класс — это ключевое слово для создания функции объекта.

Вопрос 2: Что такое объект класса?

1. У класса нет объектов.
2. Объект класса — это ключевое слово для создания класса.
3. Объект класса — это экземпляр класса.

Вопрос 3: Какое ключевое слово нужно использовать, для того, чтобы создать класс?

1. Для того чтобы создать класс, нужно использовать ключевое слово `class`.
2. Для того чтобы создать класс, нужно использовать ключевое слово `object`.
3. Для того чтобы создать класс, нужно использовать ключевое слово `class`.

Вопрос 4: Что из этого — правда о классах?

1. У классов есть свойства.
2. Класс заключается в кавычки.
3. После объявления класса, ставится тире.
4. Все из вышеперечисленного — правда.
5. Все из вышеперечисленного — не правда.

Вопрос 5: Что такое `self` в классах Python?

1. `self` — это добровольное ключевое слово в Python, которое записывается в аргумент функции класса.
2. `self` — это обязательное ключевое слово в Python, которое записывается в аргумент функции класса.
3. `self` — это обязательное ключевое слово в Python, которое записывается сразу после объявления имени класса.

Вопрос 6: Это фрагмент кода, который создает функцию

класса. Эта функция называется `attack`. Ее задача выводить сообщение “Ouch!” и сокращать количество жизней на одну единицу. Соберите фрагменты кода в правильном порядке.

1.`self.life -= 1`
2. `def attack(self):`
3.`print("Ouch!")`

ГЛАВА ДВЕНАДЦАТАЯ: МОДУЛИ И ПАКЕТЫ

Модули в Python, как и в других языка программирования — это файлы, которые содержат код. Программисты применяют модули для повторного использования одного и того же кода в программе.

Давайте узнаем о модулях и пакетах подробнее. Увидим их на примере и научимся с ними работать.

1. Введение в модули и пакеты

Модули в Python могут содержать и переменные, и функции, и другие элементы кода.

У Python есть много различных встроенных модулей, которые могут нам понадобиться в разных задачах.

Если с модулями пока все понятно, давайте узнаем, что такое пакеты модулей.

Пакеты модулей- это, попросту, папки, в которых хранится несколько модулей.

Представьте себе папку с файлами. Это и есть пакет модулей.

Довольно просто, верно? Давайте теперь научимся импортировать модули в наш код.

2. Как импортировать модуль

Для того, чтобы применить модуль, его необходимо импортировать в наш код. И как упоминалось ранее, в Python есть множество встроенных модулей.

Давайте выберем один из таких встроенных модулей — модуль

datetime, и импортируем его.

Модуль datetime и функция date

Модуль datetime делает именно то, как он и называется. Он показывает нам дату и время.

И для того, чтобы его импортировать в наш код, нам понадобится команда import и имя самого модуля.

Вот, как это выглядит на примере:

```
import datetime
```

Да, и еще кое-что: импортируя модули в Ваш код, всегда прописывайте команду import и название модуля первыми, сразу в начале кода.

Как Вы помните, программа читает код сверху вниз. И если в коде есть модули, она сразу об этом узнает, и импортирует их для корректной работы всего кода.

Вернемся к нашему модулю datetime. У него есть функция date, и она превращает числа в дату. Давайте посмотрим, как это выглядит на примере:

```
import datetime
birthday = datetime.date(2017, 6, 1)
print(birthday)
```

Запишите этот код в наш компилятор, и запустите его.

Как вы заметили, мы вводили числа через запятую. А на экране они вышли через тире.

Это произошло благодаря тому, что команда date, модуля datetime, преобразовала наши числа в дату. Получилась вот такая дата: первое июня, 2017 года.

А теперь давайте разберем этот код пошагово:

Сначала мы импортировали модуль datetime, прописав команду import и указав имя модуля.

Затем мы спустились на строчку ниже, и прописали (вызвали) модуль `datetime`, поставили точку, и прописали функцию `date`.

Затем открыли скобки и передали год, месяц и день в аргумент функции `date`.

После чего мы сохранили полученную дату в переменную `birthday` (день рождения).

И, наконец, мы спустились на строчку ниже, и вывели переменную `birthday` на экран, при помощи функции `print`.

Модуль `datetime` и функция `datetime.now`

Давайте закрепим пройденный пример еще одной функцией модуля.

Эта функция называется `datetime.now()`. И ее задача сообщать нам, какая сейчас дата и время.

Давайте напишем маленькую программу, которая сообщит нам, какой сейчас год:

```
import datetime
x = datetime.datetime.now()
print(x.year)
```

Запишите этот код в наш компилятор, и запустите его. Если Вы все сделали правильно, то программа скажет Вам, какой сейчас год.

А теперь снова разберем наш пример:

Сначала мы импортировали модуль `datetime`, прописав команду `import` и указав имя модуля.

Затем мы спустились на строчку ниже, и прописали (вызвали) модуль `datetime`, поставили точку и прописали функцию `datetime.now` с пустыми двойными скобками (Пустым аргументом).

После этого мы сохранили получившуюся дату (а получилась она благодаря функции `datetime.now`) в переменную `x`.

И, наконец, спустившись на строчку ниже, мы прописали

функцию `print`, и передали ей аргументы: нашу переменную `x` и параметр `.year`)

Работая вместе, функция `datetime.now` и параметр `.year` (год) определили текущий год. А функция `print` вывела его на экран.

Здесь хочу сказать, что `year` — это не единственный параметр, с которым мы можем работать.

Есть еще такие параметры, как: `month`, `day`, `hour`, `minute`, `second`, которые определяют соответственно: месяц, день, час, минуту и секунду.

Давайте добавим параметры “месяц” и “день” к нашему коду:

```
import datetime
x = datetime.datetime.now()
print(x.year,x.month,x.day)
```

Запишите этот код в наш компилятор, и запустите его. Как видите, теперь программа возвращает нам текущий год, месяц и сегодняшнее число.

Если с этим все понятно, вот Вам небольшая самостоятельная работа: добавьте к нашему коду оставшиеся параметры (час, минута и секунда), и запустите код.

3. Как импортировать функцию модуля

Мы можем импортировать не только модуль, но и конкретную функцию модуля.

Иногда программисты предпочитают делать именно это, импортировать не весь модуль а только конкретную его функцию.

Делают они это для того, чтобы получить больше контроля над функциями модуля в своем коде. А ещё, в случаях, когда импортировать весь модуль необязательно.

Если с этим все понятно, давайте импортируем функцию `date` модуля `datetime`:

```
from datetime import date
```

```
birthday = date(2017, 6, 1)
print(birthday)
```

Запишите этот код в компилятор, и запустите его.

Как Вы видите, этот код очень похож на тот, в котором мы импортировали весь модуль `datetime`. Тут даже вывод программы одинаковый, все та же дата: первое июня, 2017 года.

Но, в этом коде все же есть некоторые отличия. И вот они:

На первой строчке сначала ставится ключевое слово `from`, затем имя модуля, затем команда `import`, и в конце пишется имя конкретной функции модуля, которую мы хотим импортировать.

На второй строчке мы больше не вызываем модуль `datetime`. Вместо этого мы вызываем только его функцию `date`. На этом отличия заканчиваются, и далее мы передаем в аргумент функции `date` год, месяц и день.

После чего сохраняем полученную дату в переменную `birthday`.

Спускаемся на строчку ниже, и выводим переменную `birthday` с помощью функции `print`.

4. Имя функции модуля в Python

Теперь поговорим про имя функции модуля в Python и его вариации.

Дело в том, что когда мы импортируем большое количество функций модуля, одинаковые имена функций могут вызвать путаницу в коде или конфликтовать между собой.

К счастью, чтобы решить эту проблему, мы можем изменить имя функции модуля в Python, когда импортируем ее.

Для этого мы должны использовать ключевое слово `as`.

Ключевое слово `as` переводится с английского “как”. Мы, будто сообщаем программе: импортируй эту функцию как “Функция А”, а эту как “Функция Б”

Давайте переименуем функцию модуля при ее импорте:

```
from datetime import date as d
birthday = d(2017, 6, 1)
print(birthday)
```

Запишите этот код в наш компилятор, и запустите его. Давайте посмотрим, что мы сделали:

Мы взяли наш код из предыдущего задания и просто добавили в него ключевое слово “as” и новое имя “d” после команды import date.

На второй строчке мы заменили date на d. Ничего сложного, верно?

Потренируйтесь. Переименуйте функцию date по своему вкусу, и перезапустите код.

5. Как создать модуль в Python

Из знакомства с модулями мы узнали, что в Python есть большое количество встроенных модулей. Которые могут нам понадобится в разных ситуациях.

Но как и в случае с функциями, которые бывают встроенными и пользовательскими, модули тоже бывают встроенные и пользовательские.

Когда программисту необходимо создать модуль в Python, они просто записывают фрагмент рабочего кода в файл, дают файлу название, и сохраняют его под расширением .py (сокращенно от python).

Затем этот файл сохраняется в той же папке, где хранится файл с основным кодом.

Так создается модуль. Теперь, когда программисту требуется его импортировать в свой код, он просто пишет команду import и указывает название файла, в котором сохранил этот модуль.

6. Пакет модулей в Python

Как Вы уже знаете, пакет модулей в Python — это папка, где

хранится несколько модулей.

Кроме самих модулей в такой папке обязательно должен храниться файл вот с таким названием: `init.py`

Он нужен для того, чтобы программа поняла, что это пакет модулей. То есть папка, где хранятся модули.

7. Итоги двенадцатой главы

В двенадцатой главе Вы сделали следующее:

1. Узнали о модулях;
2. Научились импортировать модули;
3. Научились импортировать функции модуля;
4. Узнали о том, как создаются модули;
5. Научились присваивать имена функциям модулей;
6. Узнали о том, что такое пакеты с модулями.

8. Тест двенадцатой главы

Вопрос 1: Что из этого — правда о модулях в Python? (Множественный выбор)

1. Модули — это файлы с кодом Python.
2. У файлов модулей расширение — `.py`
3. Модули позволяют повторно использовать один и тот же код в программе.
4. Модули это папки с файлами.

Вопрос 2: Какое ключевое слово необходимо для того, чтобы импортировать модуль?

1. Для того, чтобы импортировать модуль, мы должны использовать ключевое слово `import module`.
2. Для того, чтобы импортировать модуль, мы должны использовать ключевое слово `import`.
3. Для того, чтобы импортировать модуль, мы должны использовать ключевое слово `module import`.

Вопрос 3: Расставьте код в правильной последовательности.

1. `birthday = d(2017, 6, 1)`
2. `from datetime import date as d`
3. `print(birthday)`

Вопрос 4: Когда мы используем ключевое слово `as`?

1. Когда хотим переименовать имя функции модуля.
2. Когда хотим переименовать имя модуля.
3. Когда хотим импортировать модуль.

Вопрос 5: Что из этого — правда о пакетах модулей? (множественный выбор)

1. Пакет модулей должен содержать файл `__init__.py`
2. Пакет модулей позволяет хранить модули вместе.
3. Модули одного пакета не имеют между собой ничего общего.

ГЛАВА ТРИНАДЦАТАЯ: ФИНАЛЬНЫЙ ПРОЕКТ

Добро пожаловать в наш финальный проект!

Если Вы дошли до этого момента, значит смогли постичь основы программирования на Python: типы данных и конструкции, которые включают в себя переменные, числа, строки, булеву логику, условные заявления, функции. А также основы объектно-ориентированного программирования, а именно классы, объекты и управление ими.

Все это поможет Вам без особого труда понять код нашей игры и его устройство.

А теперь приступим к делу. Ваша задача состоит в том, чтобы ознакомиться с кодом игры, который я приведу ниже, детально изучить каждый из семи этапов его создания, а затем создать свою собственную версию этой игры.

Говоря о своей собственной версии, я подразумеваю несколько вариантов.

Первый пожалуй самый легкий. В нем Вы можете взять наш код за основу и изменить его по своему вкусу. Например изменить переменные, сделать игру более сложной или более легкой.

Второй вариант предусматривает то, что Вы пойдете еще дальше и измените не только переменные, но и сценарий игры.

И наконец третий вариант для самых пытливых. В нем Вы можете создать свою собственную игру, пользуясь примером нашего кода!

И конечно же, Вы можете воспользоваться всеми вариантами по очереди, по мере роста вашего навыка в программировании и уверенности в своих силах.

Удачи!

Код игры

Важное напоминание:

Точки в начале строк указывают на отступы, которые наш компилятор будет ставить автоматически.

Все, что стоит за знаком # — это комментарии к коду, для Вашего удобства.

Между строчками кода стоят пробелы. Это только в книге. Вам не нужно делать пробелы между строчками в своем коде. Но, Вы можете ставить пробелы между блоками кода. Над каждым блоком кода находится комментарий, поэтому Вы с легкостью распознаете отдельные блоки.

```
class SoyuzDocking:
....def __init__(self):
.....self.distance = 500 # Расстояние до "Салют 7" в метрах
.....self.speed = 50 # Скорость к "Салют 7" в м/с
.....self.fuel = 100 # Количество топлива

# Сжечь топливо для замедления корабля
....def perform_burn(self, burn_amount):
.....self.speed = max(self.speed — burn_amount, 0)
.....self.fuel = max(self.fuel — burn_amount, 0)

# Обновить расстояние на основании текущей скорости
....def update_distance(self):
.....self.distance = max(self.distance — self.speed, 0)
```

```

# Проверить, состыковался-ли корабль к Салют-7
....def has_docked(self):
.....return self.distance <= 0

# Создать последовательность стыковки
docking_sequence = SoyuzDocking()

# Показать инструкции к игре
print("Добро пожаловать в симуляцию стыковки Союз Т-6!")
print("Ваша миссия — стыковка со станцией Салют-7.")
print("Вы можете управлять скоростью космического корабля
сжигая топливо.")
print("Каждая единица сожженного топлива замедляет
космический корабль на 1 м/с.")
print("Удачи экипажу!\n")

# Главный игровой цикл
while not docking_sequence.has_docked():
....print(f"Расстояние до Салют-7: {docking_sequence.distance}
метров")
....print(f"Скорость: {docking_sequence.speed} м/с")
....print(f"Топливо: {docking_sequence.fuel} кг")

# Сообщение о провале миссии в случае если закончилось
топливо
....if docking_sequence.fuel <= 0:
.....print("Кончилось топливо!")
.....break

# Запрос на активацию автопилота если расстояние до станции
менее 11 м
....if docking_sequence.distance < 11:
.....autopilot = input("До станции Салют-7 осталось менее 11
метров. Активировать режим автопилота для автоматической
стыковки? (да/нет): ")
.....if autopilot.lower() == 'да':
.....print("Автопилот активирован.")
.....break

# Запрос и ввод количества топлива, которое нужно сжечь
....burn_amount = input("Сколько сжечь топлива для снижения
скорости: ")

```

```
....burn_amount = int(burn_amount)

# Сжигание топлива и обновление расстояния до космической
станции
....docking_sequence.perform_burn(burn_amount)
....docking_sequence.update_distance()

# Завершение процесса стыковки — проверить условия и
вывести результат
if docking_sequence.distance <= 11 and docking_sequence.speed
<= docking_sequence.distance:
....print("Стыковка подтверждена. Поздравляем экипаж!")
else:
....print("Миссия провалена. Союз Т-6 не смог состыковаться с
Салют-7.")
```

Шаг 1/7: Введение в игру

Приветствую Вас, космонавт! И добро пожаловать в специальную игровую миссию!

Эта игра симулирует процесс стыковки космического корабля Союз Т-6 с орбитальной станцией Салют-7.

Ваша задача, как игрока, — успешно состыковаться с космической станцией Салют-7, управляя скоростью космического корабля и эффективно используя ограниченное количество топлива.

Игра начинается с того, что Ваш космический корабль Союз находится на расстоянии в 500 метрах от станции и движется к ней со скоростью 50 метров в секунду. Топлива у вас всего 100 килограмм.

Ваша задача — контролировать скорость космического корабля, решая, сколько топлива сжечь на каждом ходу. Каждый килограмм сожженного Вами топлива замедляет корабль ровно на один метр в секунду.

Есть два обязательных условия для победы в этой игре:

Во первых, Ваш космический корабль должен приблизиться к космической станции достаточно близко (менее 11 метров) для того, чтобы активировать режим автопилота, который и состыкует Ваш

корабль.

Во вторых, скорость Вашего космического корабля в секунду не может превышать оставшееся расстояние до космической станции, даже после активации автопилота. Иначе, Вы просто врежетесь в космическую станцию.

При этом не забывайте о запасах оставшегося топлива. Вы проигрываете, если космический корабль исчерпает запас топлива до того, как доберется до космической станции.

Помните, что исследование космоса требует точности и сообразительности. Успех вашей миссии зависит от того, сделаете ли вы правильные ходы в нужное время.

Игра написана на Python с использованием объектно-ориентированного подхода, где SoyuzDocking — это класс, который отвечает за состояния игры и методов взаимодействия с ней.

Код игры содержит цикл, который постоянно проверяет состояние игры, получая от игрока ввод и обновляя состояние игры до тех пор, пока игра не закончится.

Удачи, космонавт!

Шаг 2/7: Вспоминаем классы и объекты в Python

Как Вы уже знаете из пройденной книги, в Python класс служит шаблоном для создания объектов. У объектов есть свойства (они также называются атрибутами) и методы.

В этой игре у нас есть класс под названием SoyuzDocking. Объект этого класса представляет собой космический корабль "Союз", который должен состыковаться с космической станцией Салют-7.

Давайте разберемся с определением класса:

```
class SoyuzDocking:
    ....def __init__(self):
    .....self.distance = 500 # Расстояние до "Салют 7" в метрах
    .....self.speed = 50 # Скорость к "Салют 7" в м/с
```

```
.....self.fuel = 100 # Количество топлива
```

А теперь разберем все на части:

`class SoyuzDocking:` Эта строка начинает определение класса с именем `SoyuzDocking`.

`def __init__(self):` Это метод инициализатора, который автоматически вызывается при создании нового объекта этого класса. Он часто используется для установки начального состояния объекта.

После определения этого класса мы можем создать новый объект `SoyuzDocking` вот таким способом:

```
# Создать последовательность стыковки  
docking_sequence = SoyuzDocking()
```

В этой строке, `docking_sequence` теперь является объектом (экземпляром) класса `SoyuzDocking`, который отвечает за создание последовательности процесса стыковки.

Такой объектно-ориентированный подход позволяет нам моделировать сложные системы (например, процедуру стыковки космического корабля) при этом сохраняя наш код аккуратным и понятным.

Шаг 3/7: Детально разбираем атрибуты класса `SoyuzDocking`

В предыдущем шаге, в классе `SoyuzDocking` мы определили три атрибута в методе инициализации:

```
class SoyuzDocking:  
....def __init__(self):  
.....self.distance = 500 # Расстояние до "Салют 7" в метрах  
.....self.speed = 50 # Скорость к "Салют 7" в м/с  
.....self.fuel = 100 # Количество топлива
```

А теперь разберем детально, что представляет каждый атрибут:

`self.distance = 500:` Этот атрибут представляет расстояние в метрах до космической станции "Салют 7" в начале игры.

`self.speed = 50`: Этот атрибут представляет скорость космического корабля в метрах в секунду.

`self.fuel = 100`: Этот атрибут представляет количество доступного топлива на космическом корабле в начале игры.

Эти атрибуты критически важны для нас, поскольку именно они определяют состояние игры.

По мере прохождения игры эти значения будут меняться в зависимости от действий игрока.

Например, каждое сжигание топлива будет уменьшать атрибут топлива и изменять атрибут скорости. А атрибут расстояния будет меняться по мере приближения нашего космического корабля к космической станции.

Если с этим все понятно, давайте перейдем к четвертому шагу.

Шаг 4/7: Детально разбираем методы класса `SoyuzDocking`

В нашем классе `SoyuzDocking` есть три метода:

```
perform_burn()
update_distance()
has_docked().
```

Давайте детально их рассмотрим:

```
# Сжечь топливо для замедления корабля
....def perform_burn(self, burn_amount):
.....self.speed = max(self.speed — burn_amount, 0)
.....self.fuel = max(self.fuel — burn_amount, 0)

# Обновить расстояние исходя из текущей скорости корабля
....def update_distance(self):
.....self.distance = max(self.distance — self.speed, 0)

# Проверить, состыковался-ли корабль к Салют-7
....def has_docked(self):
.....return self.distance <= 0
```

`perform_burn(burn_amount)`: Этот метод выполняет сжигание

топлива, уменьшая скорость космического корабля и количество оставшегося топлива. Аргумент `burn_amount` указывает на то, сколько топлива сжечь.

`update_distance()`: Этот метод обновляет расстояние до космической станции, исходя из текущей скорости корабля.

`has_docked()`: Этот метод проверяет, успешно ли космический корабль состыковался с космической станцией. Если расстояние до космической станции равно 0, он возвращает `True`, указывая на успешную стыковку. В противном случае возвращает `False`.

Все эти методы определяют действия, которые наш корабль может выполнять. А также и условия, при которых корабль может пристыковаться к космической станции.

Вызывая эти методы в правильной последовательности и с правильными аргументами, мы можем смоделировать процесс пристыковки к космической станции Салют-7.

Шаг 5/7: Углубляемся в работу игрового цикла

Как Вы уже могли понять к этому моменту, игровой цикл — это конструкция управления потоком игры.

Такая конструкция, кроме самого цикла, содержит некоторые типы данных и конструкторы, которые Вам уже знакомы. Это: переменные, числа, строки, булева логика, и условные заявления. Она циклически проходит через этапы игры до тех пор, пока не будет выполнено определенное условие.

Теперь об условиях. В нашей игре, цикл будет работать до тех пор, пока наш корабль не состыкуется с "Салют-7", либо пока не кончится топливо. Или же пока мы не врежемся в космическую станцию.

Давайте теперь посмотрим на саму конструкцию:

```
# Главный игровой цикл
# 1 Выводим информацию о расстоянии, скорости и топливе
while not docking_sequence.has_docked():
    ....print(f"Расстояние до Салют-7: {docking_sequence.distance}
метров")
```

```
....print(f"Скорость: {docking_sequence.speed} м/с")
....print(f"Топливо: {docking_sequence.fuel} кг")
```

2 Сообщение о провале миссии в случае если закончилось топливо

```
....if docking_sequence.fuel <= 0:
.....print("Кончилось топливо!")
.....break
```

3 Запрос на активацию автопилота если расстояние до станции менее 11 м

```
....if docking_sequence.distance < 11:
.....autopilot = input("До станции Салют-7 осталось менее 11
метров. Активировать режим автопилота для автоматической
стыковки? (да/нет): ")
.....if autopilot.lower() == 'да':
.....print("Автопилот активирован.")
.....break
```

4 Запрос и ввод количества топлива, которое нужно сжечь

```
....burn_amount = input("Сколько сжечь топлива для снижения
скорости: ")
....burn_amount = int(burn_amount)
```

5 Сжигание топлива и обновление расстояния до космической станции

```
....docking_sequence.perform_burn(burn_amount)
....docking_sequence.update_distance()
```

6 Завершение процесса стыковки — проверить условия и вывести результат

```
if docking_sequence.distance <= 11 and docking_sequence.speed
<= docking_sequence.distance:
....print("Стыковка подтверждена. Поздравляем экипаж!")
else:
....print("Миссия провалена. Союз Т-6 не смог состыковаться с
Салют-7.")
```

Каждая итерация цикла while представляет собой новый ход в игре. На каждом ходу игра:

Выводит текущее расстояние до "Салют-7", скорость космического корабля и количество оставшегося топлива. (1)

Проверяет, не закончилось ли топливо у космического корабля. Если это так, то выводит сообщения: "Кончилось топливо!" и "Миссия провалена. Союз Т-6 не смог состыковаться с Салют-7." После чего завершает игру. (2)

Проверяет, находится ли космический корабль на расстоянии менее 11 метров от "Салют-7". И если это так, то предлагает игроку активировать автопилот. Если игрок решает активировать автопилот, то проверяет — не превышает ли текущая скорость в секунду, оставшегося расстояния до станции. Если нет, то выводит сообщения: "Автопилот активирован" и "Сстыковка подтверждена. Поздравляем экипаж!" После чего завершает игру. (3) (6)

В противном случае выводит сообщение: "Миссия провалена. Союз Т-6 не смог состыковаться с Салют-7." После чего завершает игру. (6)

Если ни одно из вышеперечисленных условий не выполняется, игра предлагает игроку очередной ход, в котором снова спрашивает игрока о том, сколько топлива сжечь. (4)

Вызывает метод `perform_burn()` объекта `SoyuzDocking` с вводом количества топлива от игрока для обновления скорости космического корабля и оставшихся запасов топлива. (5)

Вызывает метод `update_distance()` для обновления расстояния космического корабля до "Салют-7" исходя из его текущей скорости. (5)

Этот цикл является основой нашей игры, позволяя игроку взаимодействовать с ней и видеть результаты своих действий.

Цикл будет повторяться до тех пор, пока: не будет достигнут результат стыковки, или у космического корабля не закончится топливо, или корабль не врежется в космическую станцию

Шаг 6/7: Учимся принимать и обрабатывать ответ игрока

В интерактивных программах, таких, как эта игра, ввод данных от пользователя имеет огромное значение.

В нашем случае пользовательский ввод — это количество топлива, которое игрок хочет сжечь за данный игровой ход. Вот как выглядит фрагмент нашего кода, который отвечает за прием и обработку пользовательского ввода (ответа игрока):

```
burn_amount = input("Сколько сжечь топлива для снижения скорости: ")  
burn_amount = int(burn_amount)
```

Строка "Сколько сжечь топлива для снижения скорости:", выводит вопрос на экран, а функция `input()` используется для получения ответа от игрока. Затем функция ожидает, когда пользователь введет свой ответ и нажмет `enter`. Все, что пользователь вводит (кроме клавиши `enter`), возвращается функцией в виде строки.

В нашей игре мы ожидаем, что пользователь введет число, так как `burn_amount` представляет собой количество топлива. Однако, поскольку функция `input()` всегда возвращает строку, нам нужно преобразовать эту строку в число. Это делается сразу после получения ввода:

```
burn_amount = int(burn_amount)
```

Из книги Вы уже знакомы с преобразованием строковых значений. Так вот, здесь функция `int()` используется для преобразования строки в целое число.

При этом, если пользователь вводит что-то, что не может быть преобразовано в число (например, "пять" вместо 5), эта строка выдаст ошибку.

В более расширенной версии этой игры мы бы включили код для проверки ошибок, чтобы корректно обработать такой сценарий и попросить пользователя ввести ответ снова, в виде числа. Но, оставим это для будущих тем.

А пока это все для данного шага. И наша игра теперь принимает ответ от игрока, и использует его для подсчета количества топлива, которое необходимо потратить.

Шаг 7/7: Учимся выводить сообщения для игрока

Здесь все немного похоже на то, что мы уже делали в пятом шаге, однако там мы фокусировались на циклах. А здесь — на выводе сообщений для игрока. Итак приступим.

Для того чтобы игра была интересной, нам необходимо давать игроку обратную связь, как в начале игры, так и по мере ее прохождения и завершения.

Для этого, нам всегда необходимо начинать игру с подробных инструкций, что мы сейчас и сделаем.

Этот фрагмент кода выводит пять строк, которые объясняют игроку, что нужно делать. Заметили “\n” на пятой строке? Этот символ делит текст на параграфы. Он поможет нам визуально отделить нашу стартовую инструкцию от дальнейших сообщений, для удобства игрока:

```
# Показать инструкции к игре
print("Добро пожаловать в симуляцию стыковки Союз Т-6!")
print("Ваша миссия — стыковка со станцией Салют-7.")
print("Вы можете управлять скоростью космического корабля
сжигая топливо.")
print("Каждая единица сожженного топлива замедляет
космический корабль на 1 м/с.")
print("Удачи экипажу!\n")
```

Дальше все как в пятом шаге, в котором мы писали циклы.

Три сообщения, которые Вы видите ниже, сообщают игроку оперативные данные: расстояние до космической станции, текущую скорость, и количество оставшегося топлива:

```
# Главный игровой цикл
while not docking_sequence.has_docked():
    ....print(f"Расстояние до Салют-7: {docking_sequence.distance}
метров")
    ....print(f"Скорость: {docking_sequence.speed} м/с")
    ....print(f"Топливо: {docking_sequence.fuel} кг")
```

Данное сообщение появляется в случае, если у игрока закончилось топливо:

```
# Сообщение о провале миссии в случае если закончилось
топливо
....if docking_sequence.fuel <= 0:
.....print("Кончилось топливо!")
.....break
```

А это сообщение появляется в случае, если корабль приблизился на расстояние менее 11 метров от станции. Сообщение содержит вопрос об активации автопилота, варианты ответа, и еще одно сообщение, на случай если игрок активирует автопилот:

```
# Запрос на активацию автопилота если расстояние до станции
менее 11 м
....if docking_sequence.distance < 11:
.....autopilot = input("До станции Салют-7 осталось менее 11
метров. Активировать режим автопилота для автоматической
стыковки? (да/нет): ")
.....if autopilot.lower() == 'да':
.....print("Автопилот активирован.")
.....break
```

И наконец, в данном булевом конструкте есть два сообщения: первое на случай если условия для успешной стыковки выполнены. Второе напротив, на случай если такие условия не выполнены:

```
# Завершение процесса стыковки — проверить условия и
вывести результат
if docking_sequence.distance <= 11 and docking_sequence.speed
<= docking_sequence.distance:
....print("Стыковка подтверждена. Поздравляем экипаж!")
else:
....print("Миссия провалена. Союз Т-6 не смог состыковаться с
Салют-7.")
```

ГЛАВА ЧЕТЫРНАДЦАТАЯ: ЧТО ДАЛЬШЕ?

Поздравляю Вас с завершением нашей книги!

Теперь, когда Вы постигли азы программирования на Python, Вы можете приступить к самостоятельным проектам.

Так, Вы можете начать создавать свои собственные игры и

программы, наподобие тех, что мы прошли в этой книге. Экспериментировать и усложнять их код, делая Ваши игры и программы более функциональными и интересными.

Начните собирать собственную коллекцию таких проектов. Продолжайте оттачивать Ваши навыки. А когда поймете, что легко справляетесь с такими простыми программами, установите себе Visual Studio Code и начните писать полноценные приложения.

Основами Вы уже владеете, поэтому для Вас теперь открыт целый мир полезной информации, которую Вы способны понимать и применять.

“А какие именно полноценные приложения?” Спросите Вы, и будете абсолютно правы. Позвольте привести Вам некоторые примеры приложений, которые Вы можете создать с помощью Python:

Настольные приложения с графическим интерфейсом

Библиотеки Python, такие как Tkinter, PyQt или Kivy, можно использовать для создания настольных приложений с удобным графическим интерфейсом.

Разработка игр

Pygame — популярный выбор для разработки игр на Python. Он не так мощен, как Unity или Unreal Engine, но подходит для более простых 2D игр.

Веб-приложения

Фреймворки Python, такие как Django, Flask, Pyramid и др., можно использовать для создания мощных, функционально насыщенных веб-приложений.

Чат-боты

Вы можете разрабатывать интерактивные чат-боты с использованием таких библиотек Python, как ChatterBot.

Кибербезопасность и тестирование на проникновение

С помощью инструментов, таких как Scapy, nmap-python и др., можно разрабатывать приложения, связанные с безопасностью, на

Python.

Инструменты для анализа и визуализации данных

Библиотеки Python для обработки данных, такие как pandas, и библиотеки для визуализации, такие как Matplotlib и Seaborn, можно использовать для разработки приложений, которые обрабатывают, анализируют и визуализируют данные.

Приложения машинного обучения

Библиотеки, такие как TensorFlow, PyTorch, scikit-learn и др., позволяют разработчикам создавать приложения AI и машинного обучения с помощью Python.

Научные и числовые приложения

Python широко используется в научных вычислениях с помощью таких библиотек, как SciPy, NumPy и др.

Веб-скрапинг приложения

Библиотеки, такие как BeautifulSoup и Scrapy, можно использовать для разработки приложений, которые извлекают данные из веб-сайтов.

Автоматизация и написание скриптов

Простота Python делает его отличным языком для написания скриптов автоматизации. Например, вы можете автоматизировать повторяющиеся задачи, такие как перемещение файлов, анализ логов и др.

Блокчейн-приложения

Python можно использовать для разработки блокчейн-приложений, включая криптовалюты, смарт-контракты и ICO.

Обработка изображений

Библиотеки, такие как OpenCV и Pillow, можно использовать для создания приложений, которые работают с обработкой изображений.

Обработка естественного языка (NLP)

Библиотеки, такие как NLTK, spaCy и Gensim, могут помочь в разработке приложений, работающих с данными человеческого языка.

Интернет вещей (IoT)

Python можно использовать для создания приложений для IoT, с помощью Raspberry Pi и подобных устройств.

Сетевое программирование

В Python есть встроенные библиотеки, такие как asyncio, которые вы можете использовать для написания серверных, и других сетевых приложений.

Думаю этого достаточно для того, чтобы Вы поняли — насколько мощным языком программирования Вы теперь владеете. И я Вас с этим искренне поздравляю!

Выбирайте понравившееся Вам направление и начинайте практиковаться.

Удачи!

ПРИЛОЖЕНИЕ: ОТВЕТЫ К ТЕСТАМ

Тест первой главы

1. Компьютерная программа — это:

1. Набор инструкций и правил для компьютера, написанный на языке программирования.
2. Кусок кода, написанный на компьютере.
3. Загружаемая игра.

Правильный ответ — 1: Набор инструкций и правил для компьютера, написанный на языке программирования.

2. Как сделать так, чтобы компьютер вывел сообщение на экран?

1. Используя волшебное слово.
2. Используя функцию print().
3. Используя команду «Показать сообщение!»

Правильный ответ 2: Используя функцию print().

3. В каком порядке компьютер обрабатывает (считывает) код?

1. Компьютер считывает код построчно. Сверху вниз.
2. Компьютер считывает код построчно. Снизу вверх.
3. Компьютер ничего не считывает; Он все помнит наизусть.

Правильный ответ — 1: Компьютер считывает код построчно. Сверху вниз.

4. Расположите фрагменты кода так, чтобы программа отображала сообщение «Я люблю Python!»

1.)
2. (
3. "Я люблю Python!"
4. print

Правильная последовательность: print("Я люблю Python!")

Тест второй главы

1. Для чего нужны переменные?

1. Переменные нужны для хранения информации.
2. Переменные нужны для изменения информации.
3. Переменные нужны для извлечения или удаления информации.

Правильный ответ — 1: Переменные нужны для хранения информации.

2. Если имя переменной состоит из двух и более слов, Вы должны соединить их с помощью:

1. Нижнего подчеркивания.
2. Пунктирной линии.

3. Никак, это нормально — слепить все слова в одно.
4. Нужно прописать слова слитно, с большой буквы.

Правильный ответ — 1: С помощью нижнего подчеркивания.

3. Мы можем вывести переменную на экран следующим образом:

1. Используя функцию print и поместив команду print в скобки.
2. Используя функцию print и поместив значение переменной в скобки.
3. Используя функцию print и поместив имя переменной в скобки.

Правильный ответ — 3: Используя функцию print и поместив имя переменной в скобки.

4. Расположите фрагменты кода в правильной последовательности, чтобы получилась переменная, которая выводит "Илон" на экран.

1. (имя)
2. имя
3. =
4. print
5. "Илон"

Правильная последовательность:

```
имя = "Илон"  
print(имя)
```

Тест третьей главы

1. Какие типы чисел есть в Python?

1. В Python есть три типа чисел: целые, почти целые, и дробные числа.
2. В Python есть два типа чисел: целые и полуцелые.
3. В Python есть два типа чисел: Целые и дробные числа.

Правильный ответ — 3: В Python есть два типа чисел: Целые и дробные числа.

2. Как Python делит 10 на 5? (множественный выбор)

1. 10:5
2. 10/5
3. 10–5
4. 10 * 5
5. 10//5

Правильный ответ 2 и 5: 10/5 и 10//5

3. Расположите фрагменты кода так, чтобы получилось целое число 4.

1. //
2. 10*2
3. 5

Правильная последовательность: 10*2//5

Вопрос 4: Этот код выводит число 5 на экран. Но кое чего в нем не хватает. Определите, чего именно?

```
результат = 10/2  
(результат)
```

Правильный ответ: print

Тест четвертой главы

Вопрос 1: Что такое строка в Python?

1. Строка в Python — это линия, которая проходит через код.
2. Строка в Python — это простой текст, заключенный в кавычки.
3. Строка в Python — это значение с форматом целого числа.
4. Строка в Python — это значение с форматом вещественного числа.

Правильный ответ — 2: это простой текст, заключенный в кавычки.

Вопрос 2: Что нужно сделать, чтобы создать строку?

1. Нужно заключить текст в фигурные скобки.

2. Нужно заключить текст в круглые скобки.
3. Нужно заключить текст в кавычки.

Правильный ответ — 3: Нужно заключить текст в кавычки.

Вопрос 3: Расставьте код так в правильной последовательности, чтобы получить переменную со значением в виде строки. А затем вывести значение переменной на экран.

1. message
2. =
3. "Привет Илон Маск!"
4. print
5. (message)

Правильная последовательность:

```
message = "Привет Илон Маск!"  
print(message)
```

Тест пятой главы

Вопрос 1: Этот код хранит результаты сравнения двух строк, «яблоки» и «апельсины», в переменной по имени результат, а затем выводит значение переменной на экран. Но код перепутался. Расположите фрагменты кода в правильном порядке.

1. Результат
2. "апельсины"
3. (результат)
4. =
5. "яблоки"
6. print
7. !=

Правильная последовательность:

```
результат = "яблоки" != "апельсины"  
print(результат)
```

Вопрос 2: Этот код сравнивает две переменные. Какой

результат вернет код, когда мы запустим программу?

```
машина = "Tesla"  
результат = машина == "Toyota"  
print (результат)
```

Правильный ответ: False

Вопрос 3: У булевой логики есть логические значения. Их всего два. Подставьте правильные определения для каждого из двух значений.

Когда условие оказывается правдой
Когда условие оказывается неправдой

True
False

Правильный ответ:

True — когда условие оказывается правдой.
False — когда условие оказывается неправдой.

Тест шестой главы

Вопрос 1: Для чего нужны условные заявления?

1. Условные заявления нужны для создания переменной.
2. Условные заявления нужны для того, чтобы программа знала, какие инструкции и при каких условиях она должна выполнять.
3. Условные заявления нужны для создания строк.
4. Условные заявления нужны для того, чтобы ставить программе свои условия.

Правильный ответ — 2: Условные заявления нужны для того, чтобы программа знала, какие инструкции и при каких условиях она должна выполнять.

Вопрос 2: Для чего мы используем оператор if?

1. Мы используем оператор if, для того, чтобы сказать программе что делать, если условие сработало.
2. Мы используем оператор if, для того, чтобы программа смогла

сохранить файл.

3. Мы используем оператор `if`, когда еще не знаем, каким будет условие.

4. Мы используем оператор `if` для создания строки.

Правильный ответ — 1: Мы используем оператор `if`, для того, чтобы сказать программе что делать, если условие сработало.

Вопрос 3: Для чего мы используем оператор `else`?

1. Мы используем оператор `else`, для того, чтобы сказать программе, что делать, если первое условие не сработало.

2. Мы используем оператор `else`, когда не хотим использовать оператор `if`.

3. Мы используем оператор `else`, для того, чтобы программа сообщила нам свои условия.

4. Мы используем оператор `else`, когда не хотим использовать оператор `elif`.

Правильный ответ — 1: Мы используем оператор `else`, для того, чтобы сказать программе, что делать, если первое условие не сработало.

Вопрос 4: Для чего нам нужен оператор `elif`?

1. `Elif` нам не помогает. Нам помогает только `if` и `else`.

2. `Elif` помогает нам создать переменную и присвоить ей значение.

3. `Elif` помогает нам добавлять строку в наше условие.

4. `Elif` помогает нам добавлять в программу дополнительные условия и соответствующие инструкции.

Правильный ответ — 4: `Elif` помогает нам добавлять в программу дополнительные условия и соответствующие инструкции.

Вопрос 5: Расставьте операторы в правильном порядке их использования.

1. `if`
2. `else`
3. `elif`

Правильный порядок:

1. if
2. elif
3. else

Вопрос 6: Расставьте код в правильном порядке.

1. elif время < 17:
2. else:
3. if время <12:
4.print("Добрый вечер!")
5. время = 12
6.print("Доброе утро!")
7.print("Доброй ночи!")
8.print("Добрый день!")
9. elif время < 21:

Правильный порядок:

1. время = 12
2. if время <12:
3.print("Доброе утро!")
4. elif время < 17:
5.print("Добрый день!")
6. elif время < 21:
7.print("Добрый вечер!")
8. else:
9.print("Доброй ночи!")

Тест седьмой главы

Вопрос 1: Для чего программисты используют циклы?

1. Для того, чтобы зациклить код.
2. Для того, чтобы создать переменную и присвоить ей значение.
3. Для того, чтобы создать условное заявление.
4. Для того, чтобы не повторять одни и те же задачи в коде.

Правильный ответ — 4: Для того, чтобы не повторять одни и те же задачи в коде.

Вопрос 2: Как работает цикл while?

1. Цикл while работает хорошо.
2. Цикл while работает пока определенное условие действует. (То есть возвращает True)
3. Цикл while не работает по выходным.
4. Цикл while работает, пока определенное условие возвращает False.

Правильный ответ — 2: Цикл while работает пока определенное условие действует. (То есть возвращает True)

Вопрос 3: Расставьте строки кода в таком порядке, чтобы он проиграл цикл while один раз и остановил его.

1.сообщение = False
2. while сообщение == True:
3.print("Все! Хватит")
4. сообщение = True

Правильный порядок:

1. сообщение = True
2. while сообщение == True:
3.print("Все! Хватит")
4.сообщение = False

Вопрос 4: Эта программа считает от 1 до 10. Расставьте ее код в правильном порядке.

1. while число <=10:
2.print(число)
3. число = 1
4.число = число + 1

Правильный порядок:

1. число = 1
2. while число <=10:
3.print(число)
4.число = число + 1

Вопрос 5: Эта программа ведет обратный отсчет от 10 до 1. Расставьте ее код в правильном порядке.

1. while число >= 1:
2.число = число — 1
3. число = 10
4.print(число)

Правильный порядок:

1. число = 10
2. while число >= 1:
3.print(число)
4.число = число — 1

Вопрос 6: Что из этого правда о цикле for?

1. Цикл for не использует условие, как это делает цикл while. Вместо этого он повторяет фрагмент кода для каждого элемента из списка, пока список не закончится.
2. Цикл for такой же как и цикл while.
3. Цикл for не используется в Python.
4. Цикл for использует условие, как это делает цикл while. Он не повторяет фрагмент кода для каждого элемента из списка, пока список не закончится.

Правильный ответ — 1: Цикл for не использует условие, как это делает цикл while. Вместо этого он повторяет фрагмент кода для каждого элемента из списка, пока список не закончится.

Тест восьмой главы

Вопрос 1: Что делают списки в Python?

1. Списки помогают нам ничего не забыть, при покупках в магазине.
2. Списки хранят данные (значения) с разным форматом, например строки, числа и тд.
3. Списки хранят номера телефонов наших друзей.
4. Списки помогают нам сохранять значения переменных.

Правильный ответ — 2: Списки хранят данные (значения) с разным форматом, например строки, числа и тд.

Вопрос 2: Зачем списку нужен индекс?

1. Индекс не нужен списку.
2. Индекс нужен для того чтобы программа поняла о каком значении в списке идет речь.
3. Списки не имеют индекс.
4. Индекс нужен для того чтобы превратить список в кортеж.

Правильный ответ — 2: Индекс нужен для того чтобы программа поняла о каком значении в списке идет речь.

Вопрос 3: Что из этого, список?

1. ("Iron Man", "Thor", "Captain America", "Hulk")
2. ["Iron Man", "Thor", "Captain America", "Hulk"]
3. "Iron Man", "Thor", "Captain America", "Hulk"

Правильный ответ — 2: ["Iron Man", "Thor", "Captain America", "Hulk"]

Вопрос 4: Что из этого — правда о списках? (множественный выбор)

1. В начале и конце списка должны стоять квадратные скобки [].
2. Значения в списке разделяются запятой.
3. В начале и конце списка должны стоять круглые скобки.
4. Значения в списке могут иметь разный формат.
5. Значения в списке должны иметь одинаковый формат.

Правильный ответ — 1,2 и 4:

- В начале и конце списка должны стоять квадратные скобки [].
- Значения в списке разделяются запятой.
- Значения в списке могут иметь разный формат.

Тест девятой главы

Вопрос 1: Что из этого — правда о словарях? (множественный выбор)

1. Словари очень похожи на списки и кортежи. Только они не используют индекс значений. Вместо этого они используют пары состоящие из ключа и значения.
2. После создания словаря, мы больше не можем добавлять в него пары.

3. Мы можем использовать разные форматы внутри одной пары. Например ключом может быть строка а значением число. Или наоборот.

Правильный ответ — 1 и 3:

- Словари очень похожи на списки и кортежи. Только они не используют индекс значений. Вместо этого они используют пары состоящие из ключа и значения.

- Мы можем использовать разные форматы внутри одной пары. Например ключом может быть строка а значением число. Или наоборот.

Вопрос 2: Чем словари похожи на списки? (множественный выбор)

1. У них одинаковые методы добавления и обновления значений.
2. И списки и словари заключают свои значения в фигурные скобки.
3. Значения и списков и словарей могут иметь разный формат.

Правильный ответ — 1 и 3:

- У них одинаковые методы добавления и обновления значений.
- Значения и списков и словарей могут иметь разный формат.

Вопрос 3: Какой или какие из этих способов действительно создают словарь? (множественный выбор)

1. `batman = {"возраст": 36, "имя": "Bruce Wayne"}`
2. `superman = {"имя": "Kal-L"}`
3. `cities = ["New York", "Tokyo"]`
4. `years = (2030, 2040)`

Правильный ответ 1 и 2:

- `batman = {"возраст": 36, "имя": "Bruce Wayne"}`
- `superman = {"имя": "Kal-L"}`

Вопрос 4: Соберите фрагменты кода в таком порядке чтобы программа вывела сообщение “Batman”

1. `dict["герой"] = "Batman"`

2.print(сообщение)
3. dict = {}
4.сообщение = dict.pop("герой")
5. if "герой" in dict.keys():

Правильный порядок:

1. dict = {}
2. dict["герой"] = "Batman"
3. if "герой" in dict.keys():
4.сообщение = dict.pop("герой")
5.print(сообщение)

Тест десятой главы

**Вопрос 1: Что из этого — правда о функциях?
(Множественный выбор)**

1. Функции бывают встроенные и пользовательские.
2. Для того чтобы создать функцию, нам необходимо использовать ключевое слово def.
3. В функцию нужно обязательно передавать параметр.
4. Название функции нужно указывать в кавычках.

Правильный ответ — 1 и 2:

- Функции бывают встроенные и пользовательские.
- Для того чтобы создать функцию, нам необходимо использовать ключевое слово def.

**Вопрос 2: Что из этого — правда о параметрах функций?
(Множественный выбор)**

1. Параметры в функциях могут иметь только численное значение.
2. Параметры в функциях указывать не обязательно.
3. В каждой функции может быть не более двух параметров.
4. Параметры могут иметь любое значение.

Правильный ответ — 2 и 4:

- Параметры в функциях указывать не обязательно.
- Параметры могут иметь любое значение.

Вопрос 3: Соберите функции в таком порядке, чтобы программа вывела сообщение “Привет, Илон! Скажи, когда летим на Марс?”

1. сообщение("Илон", "когда летим на Марс?")
2. def поздоровайся(имя):
3.print("Скажи, " + вопрос)
4. def сообщение(имя, вопрос):
5.поздоровайся(имя)
6.print("Привет, " + имя + "!")

Правильный порядок:

1. def поздоровайся(имя):
2.print("Привет, " + имя + "!")
3. def сообщение(имя, вопрос):
4.поздоровайся(имя)
5.print("Скажи, " + вопрос)
6. сообщение("Илон", "когда летим на Марс?")

Вопрос 4: В чем разница между параметром и аргументом?

1. Разница между параметром и аргументом не критична. И они взаимно-заменяемы.
2. Аргумент это место приготовленное для будущего значения. А параметр это и есть то значение, которое пришло на место, которое приготовил аргумент.
3. Параметр это место приготовленное для будущего значения. А аргумент это и есть то значение, которое пришло на место, которое приготовил параметр.

Правильный ответ — 3: Параметр это место приготовленное для будущего значения. А аргумент это и есть то значение, которое пришло на место, которое приготовил параметр

Тест одиннадцатой главы

Вопрос 1: Что такое класс?

1. Класс — это шаблон, по которому создается объект.
2. Класс — это пример объекта.
3. Класс — это ключевое слово для создания переменной.

4. Класс — это ключевое слово для создания функции объекта.

Правильный ответ — 1: Класс — это шаблон, по которому создается объект.

Вопрос 2: Что такое объект класса?

1. У класса нет объектов.
2. Объект класса — это ключевое слово для создания класса.
3. Объект класса — это экземпляр класса.

Правильный ответ — 3: Объект класса — это экземпляр класса.

Вопрос 3: Какое ключевое слово нужно использовать, для того, чтобы создать класс?

1. Для того чтобы создать класс, нужно использовать ключевое слово `class`.
2. Для того чтобы создать класс, нужно использовать ключевое слово `object`.
3. Для того чтобы создать класс, нужно использовать ключевое слово `class`.

Правильный ответ — 3: Для того чтобы создать класс, нужно использовать ключевое слово `class`.

Вопрос 4: Что из этого — правда о классах?

1. У классов есть свойства.
2. Класс заключается в кавычки.
3. После объявления класса, ставится тире.
4. Все из вышеперечисленного — правда.
5. Все из вышеперечисленного — не правда.

Правильный ответ — 1: У классов есть свойства.

Вопрос 5: Что такое `self` в классах Python?

1. `self` — это добровольное ключевое слово в Python, которое записывается в аргумент функции класса.
2. `self` — это обязательное ключевое слово в Python, которое записывается в аргумент функции класса.
3. `self` — это обязательное ключевое слово в Python, которое

записывается сразу после объявления имени класса.

Правильный ответ — 2: `self` — это обязательное ключевое слово в Python, которое записывается в аргумент функции класса.

Вопрос 6: Это фрагмент кода, который создает функцию класса. Эта функция называется `attack`. Ее задача выводить сообщение “Ouch!” и сокращать количество жизней на одну единицу. Соберите фрагменты кода в правильном порядке.

1. `....self.life -=1`
2. `def attack(self):`
3. `....print("Ouch!")`

Правильный порядок:

1. `def attack(self):`
2. `....print("Ouch!")`
3. `....self.life -=1`

Тест двенадцатой главы

Вопрос 1: Что из этого — правда о модулях в Python? (Множественный выбор)

1. Модули — это файлы с кодом Python.
2. У файлов модулей расширение — `.py`
3. Модули позволяют повторно использовать один и тот же код в программе.
4. Модули это папки с файлами.

Правильный ответ — 1, 2 и 3:

- Модули — это файлы с кодом Python.
- У файлов модулей расширение — `.py`
- Модули позволяют повторно использовать один и тот же код в программе.

Вопрос 2: Какое ключевое слово необходимо для того, чтобы импортировать модуль?

1. Для того, чтобы импортировать модуль, мы должны использовать ключевое слово `import module`.

2. Для того, чтобы импортировать модуль, мы должны использовать ключевое слово `import`.

3. Для того, чтобы импортировать модуль, мы должны использовать ключевое слово `module import`.

Правильный ответ — 2: Для того, чтобы импортировать модуль, мы должны использовать ключевое слово `import`.

Вопрос 3: Расставьте код в правильной последовательности.

1. `birthday = d(2017, 6, 1)`
2. `from datetime import date as d`
3. `print(birthday)`

Правильная последовательность:

1. `from datetime import date as d`
2. `birthday = d(2017, 6, 1)`
3. `print(birthday)`

Вопрос 4: Когда мы используем ключевое слово `as`?

1. Когда хотим переименовать имя функции модуля.
2. Когда хотим переименовать имя модуля.
3. Когда хотим импортировать модуль.

Правильный ответ — 1: Когда хотим переименовать имя функции модуля.

Вопрос 5: Что из этого — правда о пакетах модулей? (множественный выбор)

1. Пакет модулей должен содержать файл `__init__.py`
2. Пакет модулей позволяет хранить модули вместе.
3. Модули одного пакета не имеют между собой ничего общего.

Правильный ответ — 1 и 2:

- Пакет модулей должен содержать файл `__init__.py`
- Пакет модулей позволяет хранить модули вместе.