Олег Ростиславович
Степанов

Python для начинающих

Олег Степанов

Python для начинающих

«Издательские решения»

Степанов О. Р.

Python для начинающих / О. Р. Степанов — «Издательские решения»,

ISBN 978-5-00-514576-5

Python для начинающихПрограммирование на языке Питон — популярный вид написания кода, который широко используется для решения задач совершенно разного уровня. Софт, созданный на основе данного языка, применяется компаниями и частными лицами.Просто в освоении. По сравнению с другими языками, Питон достаточно лоялен к новичкам.Можно работать прямо из браузера.Питон представляет собой востребованный язык, который используют ведущие компании, такие как Google и Pixar.

ISBN 978-5-00-514576-5

Содержание

Python для начинающих

Олег Ростиславович Степанов

ISBN 978-5-0051-4576-5
Создано в интеллектуальной издательской системе Ridero

Условный оператор

Аннотация

Во втором уроке мы впервые познакомимся с одной из фундаментально важных тем в программировании – условным оператором. Он позволяет организовать ветвление в вашей программе (выполнение одной ветки кода в зависимости от условия).

1. Повторение

На прошлом уроке мы познакомились с переменными. Переменная имеет имя и значение. Имя переменной может начинаться только с буквы и может включать в себя буквы, цифры и символ подчеркивания. Имя переменной должно отражать ее назначение.

Чтобы задать переменной значение, необходимо после знака равно (оператора присваивания) указать значение переменной.

Еще значение переменной можно получить из ввода. Для этого используем команду input (). В этом случае значение переменной задает пользователь.

2. Условный оператор

Условный оператор используется, когда некая часть программы должна быть выполнена, только если верно какое-либо условие. Для записи условного оператора используются ключевые слова if и else («если», «иначе»), двоеточие, а также отступ в четыре пробела.

if условие:
Действия, если условие верно
else:
Действия, если условие неверно

PEP 8

Отступ в 4 пробела принят в сообществе Python (PEP 8). При этом программа может работать и при других вариантах, но читать её будет неудобно. Пробелы – самый предпочтительный метод отступов. Табуляция должна использоваться только для поддержки кода, написанного с отступами с помощью табуляции. Python 3 запрещает смешивание табуляции и пробелов в отступах.

Рассмотрим пример:

```
print («Введите пароль:»)
password = input ()
if password == ’qwerty’:
print («Доступ открыт.»)
else:
print («Ошибка, доступ закрыт!»)
```

Обратите внимание: в начале условного оператора if выполняется сравнение, а не присваивание. Разница вот в чём:

Определение
Сравнение – это проверка, которая не меняет значение переменной (в сравнении может вообще не быть переменных), а присваивание – команда, которая меняет значение переменной. Для сравнения нужно использовать двойной знак равенства: ==. Также заметьте, что после else никогда не пишется никакого условия.

Другой пример:
print («Представься, о незнакомец!»)
name = input ()
if name == «Цезарь»:
print («Аве, Цезарь!»)
else:
print («Приветик.»)

В качестве условия можно использовать и другие операции отношения:
<меньше
> больше
<= меньше или равно
> = больше или равно
== равно
!= не равно

PEP 8
Все операции отношения окружаются пробелами с двух сторон.
Правильно: if bird == «Тук-тук»:
~~Неправильно~~: if bird==«Тук-тук»:

Объекты любой однородной группы можно сравнивать между собой. Подумайте над тем, как можно сравнивать, например, строки.

3. Сложное условие. Логические операции

Иногда в условном операторе нужно задать сложное условие. Для этого можно использовать логические операции and («и»), or («или») и not («не»).

Важно
Чтобы задать, что два условия должны выполняться одновременно – используем and («и»), если достаточно выполнения одного из двух вариантов (или оба сразу), то используем or («или»), а если нужно убрать какой-то вариант, то используем not («не»).

Приоритет выполнения операций:
1. not
2. and
3. or
Если нужно изменить приоритет операций или вы забыли правила – используйте скобки.

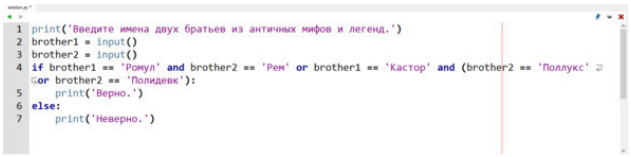
Например, вот так можно проверить, что оба условия выполнены:
print («Как называются первая и последняя буквы греческого алфавита?»)
greek_letter_1 = input ()
greek_letter_2 = input ()
if greek_letter_1 == ’альфа’ **and** greek_letter_2 == ’омега’:
print («Верно.»)
else:
print («Неверно.»)

Ниже еще несколько примеров.

print («Как греки или римляне называли главу своего пантеона – бога грома?»)
ancient_god = input ()
if ancient_god == «Зевс’ **or** ancient_god == «Юпитер’:
print («Верно.»)
else:
print («Неверно.»)

print («Введите имена двух братьев из античных мифов и легенд.»)
brother1 = input ()
brother2 = input ()
if brother1 == «Ромул» **and** brother2 == «Рем» **or** brother1 == «Кастор’ **and** (brother2 == «Поллукс’ **or** brother2 == «Полидевк»):
print («Верно.»)
else:
print («Неверно.»)

Обратите внимание, что если программу из предыдущего примера вставить в WindIDE, то часть кода условного оператора будет выходить за ограничительную красную черту среды.



PEP 8

По стандарту PEP 8 длина строки должна быть ограничена максимум 79 символами.

Предпочтительным способом переноса длинных строк является использование подразумеваемых продолжений строк Python **внутри круглых, квадратных и фигурных скобок**. Длинные строки могут быть разбиты на несколько строк, обёрнутых в скобки.

Сделайте правильные отступы для перенесённой строки. Предпочтительнее вставить перенос строки после логического оператора, но не перед ним.

Тогда представленный выше программный код может быть записан так:
print («Введите имена двух братьев из античных мифов и легенд.»)
brother1 = input ()

```
brother2 = input ()
if (brother1 == «Ромул» and brother2 == «Рем» or brother1 == «Кастор’ and
(brother2 == «Поллукс’ or brother2 == «Полидевк»)):
print («Верно.»)
else:
print («Неверно.»)
```

Рассмотрим еще несколько примеров.

```
print («Введите любые два слова, но это не должны быть „белый“
и „медведь“ разом.»)
word1 = input ()
word2 = input ()
if not (word1 == «белый» and word2 == «медведь»):
print («Верно.»)
else:
print («Неверно.»)
```

4. Вложенные условия

Важно

В команде if при выполнении условия можно выполнять более одной команды. Для этого все их необходимо выделить отступом. Такая запись называется **блоком кода**. По отступам интерпретатор определяет, какие команды исполнять при выполнении каких условий. Аналогично можно делать и для команды else.

```
print («Представься, о незнакомец!»)
name = input ()
if name == «Цезарь» or name == «Caesar’:
print («Аве, Цезарь!»)
print («Слава императору!»)
else:
print («Приветик.»)
print («Погода сегодня хорошая.»)
print («Засим – заканчиваем.»)
```

Перед последней строчкой нет отступа – это означает, что она будет выполнена в конце работы программы в любом случае. А вот две предыдущие строчки будут выполнены, только если условие if окажется ложным.

Блоки кода в Python очень гибко устроены: внутри них можно писать любой другой код, в том числе – условные операторы. Среди команд, которые выполняются, если условие if истинно («внутри if») или ложно («внутри else»), могут быть и другие условные операторы. Тогда команды, которые выполняются внутри этого внутреннего if или else, записываются с дополнительным отступом.

Изучите пример ниже. elif – это короткая запись для «else: if». Если не пользоваться короткой записью, то if пришлось бы писать на отдельной строчке и с отступом (а всё, что внутри этого if – с дополнительным отступом). Это не очень удобно, и elif избавляет от такой необходимости.

```
print («Представься, о незнакомец!»)
name = input ()
if name == «Цезарь» or name == «Caesar’:
print («Аве, Цезарь!»)
print («В честь какого бога устроим сегодня празднество?»)
god = input ()
if god == «Юпитер’:
print («Ура Громовержцу!»)
# если оказалось, что имя бога не «Юпитер’, то проверяем,
# не равно ли оно строке «Минерва’
elif god == «Минерва’:
print («Ура мудрой воительнице!»)
# следующая строка будет выполнена,
# только если имя бога не «Юпитер’ и не «Минерва’
else:
print («Бога по имени», god, «мы не знаем, но слово Цезаря – закон.»)
# эта команда будет выполнена независимо от того,
# какое имя бога ввёл пользователь, если только изначально он представился Цезарем
print («Слава императору!»)
else:
print («Приветик.»)
print («Погода сегодня хорошая.»)
print («Засим – заканчиваем.»)
```

А более простой вариант этой программы теперь попробуйте написать самостоятельно.

5. Операции над строками

Во всех примерах, которые мы рассматривали, переменные хранили строки. Мы вводили, выводили и хранили строки. Кроме уже описанных действий строки ещё можно складывать.
Давайте попробуем:

```
x = ’10»
y = ’20»
z = x + y
print (z)
```

PEP 8

И опять немного рекомендаций по оформлению (PEP 8) – ставьте пробелы вокруг знаков операций (*, +, – и т.д.)

Правильно: z = x + y

~~Неправильно~~: z = x+y

В данной программе мы задали переменным x и y значение, переменной z присвоили значение результата сложения x и y.

Результатом выполнения программы будет строка ′1020′.

Важно

Операция сложения для строк выполняет конкатенацию двух строк, то есть склеивает их содержимое вместе.

Например:

Операция «При» + «вет» в результате даст строку «Привет».

Обратите внимание, что запись: x + y = z недопустима. Оператор присваивания ожидает слева переменную, которой надо присвоить значение, а в правой части находится значение или выражение, которое надо сначала вычислить, а затем присвоить.

Мы могли сократить нашу программу и написать в таком виде:

```
x = ’10»
y = ’20»
print (x + y)
```

Результат будет такой же. Проверьте. Оператор print () сначала вычислил значение выражения x + y, а потом вывел на экран полученное значение.

А ещё такой результат можно получить вот таким образом:

```
print (’10» + ’20»)
```

Важно

Для строк так же можно выполнять умножение. Умножать можно строку на число или число на строку. Эта операция называется **дублированием**. В результате начальная строка будет повторена заданное количество раз.

Например: 3 * ′20′ то же что и ′20′ * 3, и результат будет ′202020′ и в том, и в другом случае.

Примеры использования:

```
x = ’10»
y = ’20»
print (x * 2 + y * 3)
```

Что будет на экране после запуска такой программы?

6. Команда in

Теперь рассмотрим новую команду для работы со строками – команду in.

Важно

Команда in позволяет проверить, что одна строка находится внутри другой.

Например: строка «на» находится внутри строки «сложная задача».

В таком случае обычно говорят, что одна строка является подстрокой для другой.

```
text = input ()
if ’хорош“ in text and „плох’ not in text:
print («Текст имеет положительную эмоциональную окраску.»)
elif «плох’ in text and ’хорош» not in text:
```

```
print («Текст имеет отрицательную эмоциональную окраску.»)
else:
print («Текст имеет нейтральную или смешанную эмоциональную окраску.»)
```

Первое условие окажется истинным, например, для строк «всё хорошо» и «какой хороший день», но не для «ВсЁ ХоРоШо» и не для «что-то хорошо, а что-то и плохо». Аналогично, второе условие окажется истинным для строк «всё плохо», «плохое настроение» и т. д.

Задачи

Мой Питон!

Напишите программу, которая считывает одну строку. Если это строка «My Python!», программа выводит «YES»; в противном случае программа выводит «NO»

Простые встроенные функции

Аннотация

В этом уроке мы познакомимся с типами данных, научимся работать с числами и узнаем о простейших функциях.

Повторение

На прошлом уроке мы рассмотрели условный оператор, который позволяет выполнять различные ветки кода, в зависимости от заданных условий. Научились составлять сложные условия при помощи операций not, and и or.

1. Типы данных. Числовые типы

Пока единственным типом данных, с которым мы работали, были строки. Теперь нам предстоит рассмотреть целые и вещественные числа. У каждого элемента данных, который встречается в программе, есть свой тип. (В случае Python более правильный термин – «класс объекта», но об этом мы будем говорить гораздо позже.)

Например, «привет» – это строка, а вот 15.3 – это число (дробное). Даже если данные не записаны прямо в программе, а получаются откуда-то ещё, у них есть совершенно определённый тип. Например, на место input () всегда подставляется строка, а 2 +2 даст именно число 4, а не строку «4».

Пользователь может ввести с клавиатуры какие-то цифры, но в результате input () вернёт строку, состоящую из этих цифр. Если мы попытаемся, например, прибавить к этой строке 1, то получим ошибку.

Давайте попробуем это сделать:

```
a = input ()
print (a +1)
```

Сохраните и запустите программу.

Введите любое число и посмотрите, что получится.

Ошибка возникнет потому, что в переменную а у нас попадает строка, а в функции print мы пытаемся эту строку из переменной а и число 1. Исправьте программу так, чтобы она работала.

А если нам надо работать с числами? Мы пока будем рассматривать целые и вещественные числа.

Важно

Когда речь идет о числовых данных – они записываются **без кавычек**.

А для вещественных чисел – для разделения целой и дробной части используют **точку**.

На прошлом занятии мы складывали две строки:

```
print ('10» + '20»)
```

И получали результат – строку «1020».

Давайте попробуем в этом примере убрать кавычки. В таком случае речь пойдёт уже не о строках, а о двух целых числах.

И результатом функции print (10 +20) будет целое число 30.

А если мы попробуем сложить два вещественных числа print (10.0 +20.0), то результатом будет вещественное число 30.0.

Попробуйте предположить что будет, если сложить вещественное число и целое число print (10.0 +20). Почему?

Мы выполняли сложение двух чисел внутри функции print, но мы можем переменным давать нужные значение и выполнять действия над переменными.

Давайте напишем программу, которая задаст нужные значения двум переменным (10 и 20), потом вычислит их сумму, положит это значение в третью переменную и выведет на экран полученный результат. Допишите начальные строки, чтобы программа решала поставленную задачу:

…
print (summ)

Важно

Обратите внимание, что если в качестве имени переменной для суммы взять **sum**, то оно выделяется цветом. Это означает, что такое имя знакомо среде и принадлежит какой-то функции, в качестве имени переменной его лучше не использовать.

Как складывать два числа мы научились. Еще числа можно вычитать, умножать, делить, возводить в степень, получать целую часть от деления и остаток от деления нацело. Давайте разберём эти операции на примерах.

print (30 – 10)
print (30.0 – 10)
print (3 * 3)

С вычитанием и умножением все понятно, они аналогичны сложению.

Возведение в степень обозначается двумя звёздочками **, которые должны записываться без разделителей.

print (9 ** 2)

Обратите внимание, что результат деления – всегда вещественный, даже если мы делим два целых числа, которые делятся нацело.

print (10 / 2)

Попробуйте поделить на 0. Посмотрите, как будет выглядеть ошибка деления на 0.

2. Операции над числами. Целочисленное деление

Важно

Для реализации целочисленного деления существуют два действия – деление нацело и остаток от деления нацело. Получение целой части от деления обозначается как удвоенный знак деления //, а остатка от деления нацело – %.

Давайте подробнее разберём эти операции. Что будет выведено в результате этих действий?

print (10 // 3, 10% 3)
print (10 // 5, 10% 5)

```
print (10 // 11, 10% 11)
```

Допустим, вам известны результаты a // b, a % b и число b, напишите формулу, как найти число a?

Давайте проверим вашу формулу:

```
a = 10
b = 3
print (….А сюда напишем формулу…)
```

Обратите внимание на порядок выполнения действий в вашей формуле. Целочисленное деление имеет тот же приоритет, что и обычное деление, значит, будет выполнятся раньше, чем вычитание и сложение. Для изменения приоритета выполнения операций используются скобки, все также, как и в математике.

А теперь, немного разобравшись с этими операциями, попробуйте предположить, что выведется на экран после выполнения следующего куска кода:

```
print (10 // 3, 10% 3)
print (-10 // 3, -10% 3)
```

Определите, что будет выведено на экран?

```
a = 4
b = 15
c = b / 5 * 3 – a
print (c)
```

3. Приоритет операций

Мы уже с вами изучили несколько типов операторов в языке Python:

– операторы присваивания (=, +=, -=, *= и т.д.)

– операторы сравнения (==,!=,>, <,> =, <=)

– арифметические операторы (+, -, *, //, %, **)

– логические операторы (and, or, not)

Есть и другие, мы с ними познакомимся позднее. Все эти операторы могут использоваться совместно в довольно сложных конструкциях, поэтому нужно помнить о приоритете операций, и в случае необходимости менять его при помощи скобок.

Итак, приоритет выполнения операторов в Python от наивысшего (выполняется первым) до наинизшего:

1. Возведение в степень (**)

2. Унарный минус (-). Используется для получения, например, противоположного числа.

3. Умножение, деление (* / % //)

4. Сложение и вычитание (+ -)

5. Операторы сравнения (<= <>> =)

6. Операторы равенства (==!=)

7. Операторы присваивания (=)

8. Логические операторы (not or and)

PEP 8

Если используются операторы с разными приоритетами, попробуйте добавить пробелы вокруг операторов с самым низким приоритетом. Руководствуйтесь своими собственными суждениями, но никогда не используйте более одного пробела и всегда используйте одинаковое количество пробелов по обе стороны бинарного оператора.

4. Простейшие функции

С действиями над числами определились, осталось разобраться, как получать числа из ввода. Здесь нам поможет важное новое понятие – **функция**. В математике функция из одного числа (или даже нескольких) делает другое.

Функция

В программировании (и в Python в частности): **функция** – это сущность, которая из одного (или даже нескольких) значений делает другое. При этом она может ещё и выполнять какие-то действия. Например, есть функция модуля y = |x|, аналогично в Python есть функция y = abs (x). Но функции в Python необязательно принимают только числа.

Для того, чтобы вводить числа с клавиатуры и далее работать с ними, нам необходимо найти функцию, которая из строки делает число. И такие функции есть!

Важно

Тип данных целых чисел в Python называется **int**, дробных чисел – **float**.

Одноимённые функции принимают в качестве аргумента строку и возвращают число, если в этой строке было записано число (иначе выдают ошибку):

```
a = input ()
b = int (a)
print (b +1)
```

Или можно написать даже так:

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

что будет означать – получи строку из ввода, сделай из неё целое число и результат помести в переменную a.

И тогда предыдущая программа может быть записана в виде:

```
a = int (input ())
print (a +1)
```

Но можно сократить код еще, написав вот так:

```
print (int (input ()) +1)
```

Функция int может быть применена и для получения целого числа из вещественного – в таком случае, дробная часть будет отброшена (без округления).

Например,

```
print (int (20.5 +34.1))
```

выдаст на экран число 54, хотя если сложить эти числа и не отправлять их в фунцкию int, результат будет 54.6.

В Python существует огромное количество различных функций, мы будем знакомиться с ними постепенно. Так, например, для строки можно так же определить её длину.

Важно
Длина строки – это количество символов в строке.
Для определения длины строки используется стандартная функция Python **len ()**.

На примере функции len разберемся с основными понятиями, связанными с использованием функций. Изучите код:

```
word = input ()
length = len (word)
print («Вы ввели слово длиной», length, «букв.»)
```

Важно
Когда мы в программе используем функцию, это называется **«вызов функции»**. Вызов функции устроен так: пишем имя функции – len, а затем в скобках те данные, которые мы передаём этой функции, чтобы она что-то с ними сделала. Такие данные называются **аргументами**.

В нашем примере данные в скобках должны быть строкой. Мы выбрали в качестве данных значение переменной word, которое пользователь до этого ввёл с клавиатуры. То есть значение переменной word выступает здесь в роли аргумента. А функция len выдаёт длину этой строки. Если пользователь ввёл, например, «привет», то в word оказывается равно «привет», и на место len (word) подставляется длина строки «привет», то есть 6.

Обратите внимание: каждый раз, когда мы пишем имя переменной (кроме самого первого раза – в операторе присваивания слева от знака =), вместо этого имени интерпретатор подставляет значение переменной.

Важно
Точно так же на место вызова функции (то есть имени функции и её аргументов в скобках) подставляется результат её работы – это называется **возвращаемое значение функции**.

Таким образом, функция len возвращает длину своего аргумента. input – тоже функция (отсюда скобки), она не принимает никаких аргументов, зато считывает строку с клавиатуры и возвращает её.

print – тоже функция, она не возвращает никакого осмысленного значения, зато выводит свои аргументы на экран. Эта функция может принимать не один аргумент, а сколько угодно. Несколько аргументов одной функции следует разделять запятыми.

На самом деле, функция сама по себе – это фактически небольшая программа, но об этом позже.

Функции безразлично происхождение значений, которые ей передали в качестве аргумента. Это может быть значение переменной, результат работы другой функции или записанное прямо в коде значение:

print («Это слово длиной», len (’абракадабра’), «букв.»)

Обратите внимание, что в предыдущем примере значение переменной word вообще никак не изменилось от вызова функции len. С другой стороны, вызов функции может стоять где угодно, не обязательно сразу класть возвращаемое значение в переменную.

Как есть функция int, которая пытается сделать из того, что ей передали, целое число, так же есть и функция str, которая возвращает строку из тех данных, что в нее передали.

print (str (10) + str (20)) *# выведет «1020»*
print (int (’10») + int (’20»)) *# выведет 30*

Каждый раз, когда вы пишете программу, важно понимать, какой тип имеет каждое значение и каждая переменная.

5. Обмен значениями переменных

Мы изучили операции с различными типами данных.

Давайте попробуем написать программу, которая поменяет местами содержимое переменных a и b. Пусть есть такой код:

```
a = 3
b = 5
…
…
print (a)
print (b)
```

Что надо вписать в пропущенные места, чтобы в a лежало 5, а в b лежало 3? При этом, числами 3 и 5 пользоваться нельзя.

Как один из вариантов – можно использовать дополнительную переменную:

```
a = 3
b = 5
c = a
a = b
b = c
print (a)
print (b)
```

А теперь попробуйте написать вариант без дополнительной переменной, через сумму двух чисел.

Но нам с вами очень повезло, что мы изучаем язык Python, потому что он и поддерживает более простой вариант записи:

```
a = 3
b = 5
a, b = b, a
print (a)
print (b)
```

Значения переменных, которые расположены справа от знака «присвоить», в указанном порядке помещаются в переменные слева, в порядке их указания.

Так, используя множественное присваивание, можно задавать нескольким переменным одно значение:

a = b = c = 5

Задачи
Проверка на четность

Напишите программу, которая принимает на вход число, а затем выводит «ДА», если оно четное, и «НЕТ», если нечетное.

Проверка на четность – 2

Придумаем новое условие для «нашей четности». Пусть число «четно», если его первая цифра четная,

и, соответственно, «нечетно», если первая цифра нечетная.

Программа должна принять на вход трехзначеное число и выводить «Четное» или «Нечетное».

Знакомство с циклом while

Аннотация

В этом уроке мы познакомимся с оператором цикла while. Цикл позволяет организовывать многократное повторение одних и тех же действий. Мы также сакцентируем внимание на том, что в одной и той же строчке программы на разных итерациях цикла переменные могут иметь разное значение.

1. Цикл while

Сегодня мы научимся повторять заданные действия несколько раз. Для этого существуют операторы циклов. Мы разберем оператор цикла while. Он выполняет блок кода, **пока истинно** какое-то условие.

Напомним, условный оператор if проверяет условие и, в зависимости от того, истинно оно или ложно, выполняет либо не выполняет следующий записанный с отступом блок. После этого программа в любом случае выполняется дальше (там ещё может быть elif или else, но сути это не меняет).

Важно

Оператор while («пока») тоже проверяет условие и тоже в случае его истинности выполняет следующий блок кода (**«тело цикла»**). Однако после выполнения этого блока кода выполняется не то, что идёт после него, а снова проверяется условие, записанное после while.

Ведь при выполнении тела цикла значения каких-то переменных могли измениться – в результате условие цикла может уже не быть истинным. Если условие всё ещё истинно, тело цикла выполняется снова. Как только условие цикла перестало выполняться (в том числе если оно с самого начала не было выполнено), программа идёт дальше – выполняются команды, записанные после тела цикла.

Условие цикла записывается как и для if – с помощью операций отношения (>,> =, <, <=,!=, ==). Сложные условия можно составлять с помощью логических операций not, and, or.

Действия, расположенные в теле цикла (блок кода), записываются со смещением вправо на 4 пробела относительно начала слова while. Переменные, входящие в условие, должны на момент проверки условия цикла иметь значения.

while условие:
блок кода (тело цикла)

Важно

Один шаг цикла (выполнение тела цикла) ещё называют **итерацией**.

Используйте цикл while всегда, когда какая-то часть кода должна выполниться несколько раз – причём невозможно заранее сказать, сколько именно.

Давайте посмотрим программу, в которой цикл будет выполняться пока не введут число меньше 0:

```
number = int (input ())
while number> 0:
print («Вы ввели положительное число! Вводите дальше.»)
number = int (input ())
```

```
print («Так-так, что тут у нас…»)
print («Вы ввели отрицательное число или ноль. Всё.»)
```

Разберёмся, как будет работать эта программа.

Сначала выполняется первая строчка: number = int (input ()) – пользователь вводит целое число. (*Мы предполагаем, что пользователь действительно ввёл число, и программа не вылетела с ошибкой.*) Предположим, он ввёл число 10. Оно записано в переменной number.

Выполняется вторая строчка: while number> 0: – «пока number> 0» – здесь проверяется, выполнено ли условие number> 0. Поскольку мы предположили, что number в этот момент равно 10, то да, условие выполнено, поэтому дальше выполняется блок, записанный с отступом – тело цикла.

Третья строчка программы выводит на экран строку, тут всё понятно.

Четвёртая строчка вновь считывает с клавиатуры число и сохраняет его в переменную number. Пусть пользователь ввёл 2.

Когда выполнение программы доходит до конца тела цикла, происходит возврат к заголовку цикла (второй строчке программы) и повторная проверка условия. Поскольку 2> 0, снова выполняется тело цикла.

Третья строчка снова выводит на экран сообщение, четвёртая строчка снова считывает число (пусть это будет число 3), пятая строчка снова выводит на экран сообщение…

Закончив тело цикла, опять проверяем условие в заголовке. number равно 3, 3> 0, поэтому продолжаем.

Третья строчка опять выводит на экран сообщение, четвёртая строчка опять считывает число. Пусть теперь это будет —1 – обратите внимание, переменная number на каждой итерации цикла приобретает новое значение! Пятая строчка опять выводит на экран сообщение…

Вновь вернувшись на вторую строчку, получаем, что —1> 0 – ложно. Поэтому цикл завершается, тело цикла больше не выполняется, прыгаем сразу на следующую после цикла строчку программы – шестую. Она выводит последнее сообщение.

Всё.

2. Составной оператор присваивания

Напомним, что в операторе присваивания одно и то же имя переменной может стоять и справа (в составе какого-то выражения), и слева. В этом случае сначала вычисляется правая часть со старым значением переменной, после чего результат становится новым значением этой переменной. Ни в коем случае не воспринимайте такой оператор присваивания как уравнение!

```
number = int (input ()) # например, 5
number = number +1 # тогда здесь number становится равным 6
print (number)
```

Важно

Для конструкций вида number = number +1, также существует сокращённая форма записи оператора присваивания: number += 1. Аналогично оператор x = x + y можно записать как x += y, оператор x = x * y – как x *= y, и так для любого из семи арифметических действий.

3. Сигнал остановки

Рассмотрим такую задачу: пользователь вводит числа. Пусть это будут цены на купленные в магазине товары, а наша программа – часть программного обеспечения кассового аппарата. Ввод «—1» – сигнал остановки. Нужно сосчитать сумму всех введённых чисел (сумму чека).

Поскольку требуется повторить нечто (ввод очередной цены) неизвестное количество раз, потребуется цикл while. Нам понадобится как минимум две переменные: price для цены очередного товара и total для общей суммы.

Если бы мы знали точно, что пользователю надо купить ровно три товара, то цикл (и ввод «—1» как условие его прерывания) был бы не нужен. Тогда программа могла бы выглядеть так:

```
total = 0
price = float (input ())
total = total + price
price = float (input ())
total = total + price
price = float (input ())
total = total + price
print («Сумма введённых чисел равна', total)
```

Обратите внимание: мы назвали переменные осмысленно. Это очень облегчит жизнь программисту, который будет читать наш код позже – даже если это будете вы сами неделю спустя. Однако интерпретатор Python к этому факту совершенно равнодушен. Чтобы значения переменных соответствовали названиям и тому смыслу, который мы в них закладываем, нужно поддерживать переменные в актуальном состоянии. И только вы, программист, можете это сделать. С переменной price всё более или менее понятно: её значение обновляется при считывании с клавиатуры на каждой итерации цикла, как это делалось во многих других задачах. total сначала равно нулю: до начала ввода цен их сумма, конечно, ноль. Однако значение переменной total устаревает каждый раз, когда пользователь вводит цену очередного товара. Поэтому нам нужно прибавить к значению total только что введённую цену, чтобы эта переменная по-прежнему обозначала сумму цен всех купленных товаров.

Если бы мы хотели сократить запись, можно было бы организовать цикл, который выполнился бы ровно три раза. Для этого нам потребуется переменная счетчик, которая внутри цикла будет считать каждую итерацию цикла. А условием выхода – поставим выполнение нужного количества итераций:

```
count = 0
total = 0
while count <3:
price = float (input ())
total = total + price
count = count +1
print («Сумма введённых чисел равна', total)
```

Обратите внимание, что total и count должны обнуляться до цикла.

Однако у нас в задаче количество товаров неизвестно, поэтому понадобится цикл до ввода сигнала остановки (» —1»). С учётом сказанного выше программа будет выглядеть так:

```
total = 0
print («Вводите цены; для остановки введите -1.»)
price = float (input ())
while price> 0:
total = total + price # можно заменить на аналогичное total += price
price = float (input ())
print («Общая стоимость равна', total)
```

4. Подсчет количества элементов, удовлетворяющих условию

А теперь рассмотрим ещё одну задачу.

Пользователь вводит целые числа. Ввод чисел прекращается, если введено число 0. Необходимо определить сколько чисел среди введенных оканчивались на 2 и были кратны числу 4. Теперь нам надо проверять последовательность чисел.

Для каждого введенного числа надо делать проверку, соответствует ли оно условию. Если оно подходит под условие, то увеличиваем счётчик таких чисел.

И уже после цикла, когда остановился ввод чисел – выводим результат – посчитанное количество нужных чисел.

```
count = 0
number = int (input ())
while number!= 0:
if number % 10 == 2 and number % 4 == 0:
count += 1
number = int (input ())
print («Количество искомых чисел:», count)
```

Обратите внимание, до цикла необходимо задать начальное значение для переменной count. Ведь когда придет первое подходящее под условие число, у нас count будет увеличиваться на 1, относительно предыдущего значения. А значит, это значение должно быть задано.

Давайте посмотрим, как будет работать эта программа для последовательности чисел: 12, 3, 32, 14, 0.

ШАГ	Действие	Пояснение	Цикл
1	count = 0	count = 0	
2	number = int(input())	number = 12	
3	while number != 0:	12 != 0 (верно)	1 итерация
4	if number % 10 == 2 and number % 4 == 0:	12 % 10 == 2 and 12 % 4 == 0 (верно)	заходим в if
5	count += 1	count = 1	
6	number = int(input())	number = 3	
7	while number != 0:	3 != 0 (верно)	2 итерация
8	if number % 10 == 2 and number % 4 == 0:	3 % 10 == 2 and 3 % 4 == 0 (ложно)	пропускаем if
9	number = int(input())	number = 32	
10	while number != 0:	32 != 0 (верно)	3 итерация
11	if number % 10 == 2 and number % 4 == 0:	32 % 10 == 2 and 32 % 4 == 0 (верно)	заходим в if
12	count += 1	count = 2	
13	number = int(input())	number = 14	
14	while number != 0:	14 != 0 (верно)	4 итерация
15	if number % 10 == 2 and number % 4 == 0:	14 % 10 == 2 and 14 % 4 == 0 (ложно)	пропускаем if
16	number = int(input())	number = 0	
17	while number != 0:	0 != 0 (ложно)	ВЫХОД из цикла
18	print('Количество искомых чисел:',count)	Вывод вычисленного count	

5. Поиск минимума и максимума

Очень часто в задачах приходится использовать различные статистические алгоритмы: поиск максимума, минимума, среднего значения, медианы и моды чисел, главный из которых – определение максимального и минимального значений на множестве данных.

Рассмотрим алгоритм в общем виде:

1. заведем отдельную переменную для хранения максимума и минимума. В качестве начального значения можно задать:

– заведомо малое для анализируемых данных значения, для максимума – это будет очень маленькое число, например, если мы вычисляем максимальный балл за экзамен, то можно взять maximum = 0, тогда гарантированно произойдет замена максимума. Минимуму же наоборот присваивается заведомо большое значение

– первый элемент данных

2. в теле цикла каждый подходящий элемент данных обрабатывается операторами по принципу:

– если текущий элемент больше максимума, меняем максимум

– если текущий элемент меньше минимума, заменяем минимум

Рассмотрим пример. Витя анализировал список литературы и решил, что хочет начать с самой большой по количеству страниц книги. Напишем программу, которая поможет Вите определить сколько страниц ему предстоит прочитать. Витя последовательно вводит количество страниц каждой книги из списка, а окончанием ввода служит ввод любого отрицательного числа.

```
biggest_book = 0
n = int (input ())
while n> 0:
if n> biggest_book:
biggest_book = n
n = int (input ())
print (biggest_book)
```

Так как книга не может содержать в себе 0 страниц, для значения максимума мы можем взять 0.

После этого Витя начинает вводить количество страниц, например, он вводит 148. 148> 0 – условие цикла выполняется и мы переходим к операции сравнения. На данном шаге 148> 0, значит biggest_book = 148. Снова считываем число.

Предположим теперь введено 120. 120> 0 – продолжаем работать в цикле. 120> 148 – условие не выполняется, переходим к вводу новых данных, biggest_book все еще равен 148.

В этот раз Витя ввел 486, мы заходим в цикл 486> 148, производим замену biggest_book = 486. Продолжаем ввод. И так далее до тех пор, пока не будет введено отрицательное число.

Задачи

Пароль 123456

Напишите программу, которая запршивает строчку с паролем. Если пароль короче 8 символов, то выводите «Короткий», если пароль содержит «123», то – «Простой». Запрашивайте парль до тех пор, пока

он не пройдет проверку, а затем выводите «ОК».

Знакомство с циклом for

Аннотация

В уроке рассматриваются именованные аргументы функции print, специальные символы в строке и конструкция «for… in range (…):».

1. Именованные аргументы функции print

Мы уже пользовались тем, что функция print при выводе разделяет аргументы пробелами, а в конце переходит на новую строчку.

Часто это удобно. Но что, если от этого нужно избавиться? В примере ниже пробелы появляются не только после двоеточий (что хорошо), но и перед запятой (что плохо).

```
measures = 7
cuts = 1
print («Количество отмеров:», measures,», количество отрезов:», cuts)
# выведет: «Количество отмеров: 7, количество отрезов: 1»
```

Важно

Для такой тонкой настройки вывода у функции print существуют **необязательные именованные аргументы.**

(Такие удобные инструменты бывают и у других функций, мы познакомимся с ними позже).

Обычно при вызове функции мы пишем имя функции, а затем в скобках её аргументы через запятую. Стандартный способ сообщить функции, что и с какими аргументами делать (например, какой из аргументов функции print вывести первым, какой вторым и т. д.), – это передать аргументы в нужном порядке. Например, функция print выводит аргументы именно в том порядке, в котором их ей передали. Однако есть и другой способ – именованные аргументы. Чтобы при вызове функции передать ей именованный аргумент, нужно после обычных аргументов написать через запятую имя аргумента, знак «=» и значение аргумента.

Важно

Функция print наряду с другими аргументами может (вместе или по отдельности) принимать вот таких два аргумента: **sep** – разделитель аргументов (по умолчанию пробел) и **end** – то, что выводится после вывода всех аргументов (по умолчанию – символ начала новой строки).

В частности, если end сделать пустой строкой, то print не перейдёт на новую строчку, и следующий print продолжит вывод прямо на этой же строке.

```
print («При»)
print («вет!»)
# эти две строки кода выведут «При» и «вет!» на отдельных строках
print («При», end=«»)
print («вет!») # эти две строки кода выведут «Привет!»
print («Раз», «два’, «три») # выведет «Раз два три»
```

print («Раз», «два’, „три“, sep=' – ») *# выведет «Раз – два – три»*

Обратите внимание: знак «=» здесь не выполняет никакого присваивания, переменных end и sep не появляется.

PEP 8

Не используйте пробелы вокруг знака «=», если он используется для обозначения именованного аргумента.
Правильно:

print («При», end=«»)

Задачи
Квадраты по порядку

Выведите в столбик квадраты натуральных чисел по порядку: 1, 4, 9, …, n**2, где n – введенное пользователем число.

Пирамида

При помощи пробелов и «*» выведите пирамиду заданной высоты. Верхушка – 1 *, Основание – n *.

Гипербола

Выведите таблицу значений аргумента x и значения f (x) = 1 / x, где x принадлежит натуральным числам от 1 до n.

Вложенные циклы

Аннотация

В этому уроке мы рассмотрим вложенные циклы, позволяющие запустить цикл внутри циклического оператора. Приведем несколько примеров вложенности разных циклов, а также применение операторов break и continue со вложенными циклами

1. Вложенные циклы. Принцип работы

Часто бывают ситуации, когда один и тот же набор действий необходимо выполнить несколько раз для каждого повторяющегося действия. Например, мы уже несколько раз с вами сталкивались с задачами, когда программа получает от пользователя данные до сигнала остановки, для этого используется цикл. А теперь представьте, что после ввода данных или числа с ними надо сделать какие-либо действия, которые тоже требуют цикла (например, вычислить факториал), тогда нам нужен еще один цикл, внутри первого.

Вложенные циклы

Циклы называются **вложенными** (т.е. один цикл находится внутри другого), если внутри одного цикла во время каждой итерации необходимо выполнить другой цикл. Так для каждого витка внешнего цикла выполняются все витки внутреннего цикла. Основное требование для таких циклов – чтобы **все** действия вложенного цикла располагались внутри внешнего.

При использовании вложенных циклов стоит помнить, что изменения, внесенные внутренним циклом в какие-либо данные, могут повлиять и на внешний цикл.

Давайте рассмотрим следующую задачу: необходимо вывести в строку таблицу умножения для заданного числа. Задача решается так:

```
k = int (input ())
for i in range (1, 10):
print (i, «*», k, '=», k * i, sep=«», end='\t')
```

А если нам нужно вывести таблицу умножения для всех чисел от 1 до k?

Очевидно, что в этом случае предыдущую программу нужно повторить k раз, где вместо k будут использоваться числа от 1 до k (включительно).

Эту задачу можно записать двумя циклами, где для каждого значения внешнего цикла будут выполняться все значения внутреннего цикла.

Программа будет выглядеть так:

```
k = int (input ())
for j in range (1, k +1):
for i in range (1, 10):
print (i, «*», j, '=», j * i, sep=«», end='\t')
print ()
```

Проанализируем работу данной программы. Выполнение программы начинается с внешнего цикла. Итератор j внешнего цикла for меняет свое значение от начального (1) до конечного (k). Обратите внимание, чтобы включить число k в рассматриваемый диапазон,

в заголовке цикла указавается промежуток от 1 до k+1. Затем циклически выполняется следующее:

1. Проверяется условие j <k+1.

2. Если оно соблюдается, то выполняется оператор в теле цикла, т.е. выполняется внутренний цикл.

– Итератор i внутреннего цикла for будет изменять свои значения от начального (1) до конечного (10), не включая 10.

Затем циклически выполняется следующее:

– проверяется условие i <10;

– если оно удовлетворяется, то выполняется оператор в теле цикла, т. е. оператор print (i, «*», j, '=», j*i, sep=«», end='\t'), выводящий на экран строку таблицы умножения в соответствии с текущими значениями переменных i и j;

– затем значение итератора i внутреннего цикла увеличивается на единицу, и оператор внутреннего цикла for проверяет условие i <10. Если условие соблюдается, то выполняется тело внутреннего цикла при неизменном значении итератора внешнего цикла до тех пор, пока выполняется условие i <10;

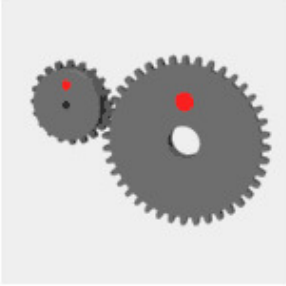
– если условие i <10 не удовлетворяется, т. е. как только i станет равен или больше 10, оператор тела цикла не выполняется, внутренний цикл завершается и управление в программе передается за пределы оператора for внутреннего цикла, т. е. выполняется перевод строки, вызванный использованием функции print () (строка 5), а затем возвращается к оператору for внешнего цикла.

3. Значение итератора внешнего цикла j увеличивается на единицу, и проверяется условие j <k+1. Если условие не соблюдается, т. е. как только j станет больше k, оператор тела цикла не выполняется, внешний цикл завершается и управление в программе передается за пределы оператора for внешнего цикла, т. е. в данном случае программа завершает работу.

Таким образом, на примере печати таблицы умножения показано, что при вложении циклов внутренний цикл выполняется полностью от начального до конечного значения параметра, при неизменном значении параметра внешнего цикла. Затем значение параметра внешнего цикла изменяется на единицу, и опять от начала и до конца выполняется вложенный цикл. И так до тех пор, пока значение параметра внешнего цикла не станет больше конечного значения, определенного в операторе for внешнего цикла.

2. Графическое представление вложенных циклов

Работу циклов также можно сравнить с вращением связанных шестерёнок разного размера:



Внешний цикл – это как бы большая шестерёнка, за один свой оборот (виток цикла), внешний цикл заставляет вращаться вложенный цикл (меньшую шестерёнку) несколько раз.

Обратите внимание, что такая иллюстрация точна в случае, если число повторов вложенного цикла не зависит от того какой именно (1-ый, n-ый или иной) виток делает внешний цикл, а так бывает не всегда.

3. Оператор break и continue во вложенных циклах

Рассмотрим другую задачу: представьте, что необходимо распечатать все строки таблицы умножения для чисел от 1 до 10, кроме строки для числа k.

Тогда нам нужно будет пропустить выполнение внутреннего цикла, когда придет k-ая строка.

Это можно сделать через оператор continue, который просто прервет выполнение данного витка цикла и перейдет к следующей итерации цикла:

```
k = int (input ())
for j in range (1, 10):
if j == k:
continue
for i in range (1, 10):
print (i, «*», j, '=», j * i, sep=«», end='\t')
print ()
```

Важно

Обратите внимание, если оператор break или continue расположен внутри вложенного цикла, то он действует именно на вложенный цикл, а не на внешний. Нельзя выскочить из вложенного цикла сразу на самый верхний уровень.

А теперь попробуйте вывести всю таблицу умножения кроме столбца k.

Вот еще одна программа, которая использует вложенные циклы и оператор break. Она учит пользователя вводить числа палиндромы – программа выполняется до тех пор, пока не будет введено число палиндром:

```
print («Тренажер по вводу палиндрома:»)
while True:
print («Введите число палиндром:»)
number = n = int (input ())
reverse = 0
while n> 0:
reverse = reverse * 10 + n % 10
n //= 10
if number == reverse:
print («Вы ввели палиндром! Программа остановлена.»)
break
print («Введенное число не палиндром, попробуйте еще раз.»)
```

Задачи
Битовые операции

Есть перменная, которая изначальна равна 1. Вводится n строчек, в каждой из которых есть одна команда: «Меняй» – изменить текущее значение на противоположное

(0 на 1 или 1 на 0), «Отдыхай» – ничего не меняй. Выводи значение переменной на каждую команду.

Простой пример

Ученик проверяет себяя при помощи калькулятора. Он вводит два числа через пробел, затем вводит их сумму. Если сумма посчитана правильно, то программа не должна ничего выводить. а если нет – писать «Ошибка, ответ – …". Так до тех пор, пока ученик не напишет «Это были простые примеры».

Множества

Аннотация

В этом уроке мы обсудим множества Python. Этот тип данных аналогичен математическим множествам, он поддерживает быстрые операции проверки наличия элемента в множестве, добавления и удаления элементов, а также операции объединения, пересечения и вычитания множеств.

1. Объекты типа set

Мы написали уже много программ, работающих с данными, количество которых неизвестно на момент написания программы. Теперь было бы здорово уметь хранить в памяти неизвестное на момент написания программы количество данных. В этом нам помогут так называемые **коллекции** – специальные типы данных, которые «умеют» хранить несколько значений под одним именем. Первая из коллекций, с которой мы познакомимся, называется **множество**.

Множество

Множество – это составной тип данных, представляющий собой несколько значений (элементов множества) под одним именем. Этот тип называется **set** – не создавайте, пожалуйста, переменные с таким именем! Чтобы задать множество, нужно в фигурных скобках перечислить его элементы.

Здесь создается множество из четырех элементов (названий млекопитающих), которое затем выводится на экран:

```
mammals = {'cat', 'dog', 'fox', 'elephant'}
print (mammals)
```

Введите этот код в Python и запустите программу несколько раз. Скорее всего, вы увидите разный порядок перечисления млекопитающих – это происходит потому, что элементы во множестве Python не упорядочены. Это позволяет быстро выполнять операции над множествами, о которых мы скоро поговорим чуть позже.

Важно

Для **создания пустых множеств** обязательно вызывать **функцию set**: empty = set ()

Обратите внимание: элементами множества могут быть строки или числа. Возникает вопрос: а может ли множество содержать и строки, и числа? Давайте попробуем:

```
mammals_and_numbers = {'cat', 5, 'dog', 3, 'fox', 12, 'elephant', 4}
print (mammals_and_numbers)
```

Как видим, множество может содержать и строки, и числа, а Python опять выводит элементы множества в случайном порядке. Заметьте, что если поставить в программе оператор вывода множества на экран несколько раз, не изменяя множество, порядок вывода элементов не изменится.

Может ли элемент входить во множество несколько раз? Это было бы странно, так как совершенно непонятно, как отличить один элемент от другого. Поэтому множество содержит каждый элемент только один раз. Следующий фрагмент кода это демонстрирует:

```
birds = {'raven', 'sparrow', 'sparrow', 'dove', 'hawk', 'falcon'}
print (birds)
```

Важно

Итак, у множеств есть три ключевые особенности:

– Порядок элементов во множестве не определён.

– Элементы множеств – строки и/или числа.

– Множество не может содержать одинаковых элементов.

Выполнение этих трёх свойств позволяет организовать элементы множества в структуру со сложными взаимосвязями, благодаря которым можно быстро проверять наличие элементов в множестве, объединять множества и так далее. Но пока давайте обсудим эти ограничения.

2. Операции над одним множеством

Простейшая операция – **вычисление числа элементов** множества. Для это служит функция **len**. Мы уже встречались с этой функцией раньше, когда определяли длину строки:

```
my_set = {'a', 'b', 'c'}
n = len (my_set) # => 3
```

Далее можно **вывести элементы множества** с помощью функции print:

```
my_set = {'a', 'b', 'c'}
print (my_set) # => {'b', 'c', 'a'}
```

В вашем случае порядок может отличаться, так как правило упорядочивания элементов во множестве выбирается случайным образом при запуске интерпретатора Python.

Очень часто необходимо обойти все элементы множества в цикле. Для этого используется цикл for и оператор in, с помощью которых можно перебрать не только все элементы диапазона (как мы это делали раньше, используя range), но и элементы множества:

```
my_set = {'a', 'b', 'c'}
for elem in my_set:
print (elem)
```

такой код выводит:

```
b
a
c
```

Однако, как и в прошлый раз, в вашем случае порядок может отличаться: заранее он неизвестен. Код для работы с множествами нужно писать таким образом, чтобы он правильно работал при любом порядке обхода. Для этого надо знать два правила:

– Если мы не изменяли множество, то порядок обхода элементов тоже не изменится.

– После изменения множества порядок элементов может измениться произвольным образом.

Чтобы **проверить наличие элемента** во множестве, можно воспользоваться уже знакомым оператором in:

```
if elem in my_set:
print («Элемент есть в множестве’)
else:
print («Элемента нет в множестве’)
```

Выражение elem in my_set возвращает True, если элемент есть во множестве, и False, если его нет. Интересно, что эта операция для множеств в Python выполняется за время, не зависящее от мощности множества (количества его элементов).

Добавление элемента в множество делается при помощи add:

```
new_elem = ’e’
my_set.add (new_elem)
```

add – это что-то вроде функции, «приклеенной» к конкретному множеству. Такие «приклеенные функции» называются **методами**.

Таким образом, если в коде присутствует имя множества, затем точка и еще одно название со скобками, то второе название – имя метода. Если элемент, равный new_elem, уже существует во множестве, то оно не изменится, поскольку не может содержать одинаковых элементов. Ошибки при этом не произойдёт. Небольшой пример:

```
my_set = set ()
my_set.add (’a’)
my_set.add (’b’)
my_set.add (’a’)
print (my_set)
```

Данный код три раза вызовет метод add, «приклеенный» к множеству my_set, а затем выведет либо {’a’, ’b’}, либо {’b’, ’a’}.

С **удалением элемента** сложнее. Для этого есть сразу три метода: **discard** (удалить заданный элемент, если он есть во множестве, и ничего не делать, если его нет), **remove** (удалить заданный элемент, если он есть, и породить ошибку KeyError, если нет) и **pop**. Метод pop удаляет некоторый элемент из множества и возвращает его как результат. Порядок удаления при этом неизвестен.

```
my_set = {’a’, ’b’, ’c’}

my_set. discard (’a’) # Удалён
my_set. discard (’hello’) # Не удалён, ошибки нет
my_set.remove (’b’) # Удалён
print (my_set) # В множестве остался один элемент ’c’
my_set.remove (’world’) # Не удалён, ошибка KeyError
```

На первый взгляд, странно, что есть метод remove, который увеличивает количество «падений» вашей программы. Однако, если вы на 100 процентов уверены, что элемент должен быть в множестве, то лучше получить ошибку во время отладки и исправить её, чем тратить время на поиски при неправильной работе программы.

Метод pop удаляет из множества случайный элемент и возвращает его значение:

```
my_set = {’a’, ’b’, ’c’}
print («до удаления:», my_set)
elem = my_set. pop ()
print (’удалённый элемент:», elem)
print («после удаления:», my_set)
```

Результат работы случаен, например, такой код может вывести следующее:

```
до удаления: {’b’, ’a’, ’c’}
удалённый элемент: b
после удаления: {’a’, ’c’}
```

Если попытаться применить pop к пустому множеству, произойдёт ошибка KeyError.
Очистить множество от всех элементов можно методом **clear**:
my_set.clear ()

3. Операции над двумя множествами

Есть четыре операции, которые из двух множеств делают новое множество: объединение, пересечение, разность и симметричная разность.

Объединение двух множеств включает в себя все элементы, которые есть хотя бы в одном из них. Для этой операции существует метод **union**:
union = my_set1.union (my_set2)

Или можно использовать оператор |:
union = my_set1 | my_set2

Пересечение двух множеств включает в себя все элементы, которые есть в обоих множествах:

intersection = my_set1.intersection (my_set2)

Или аналог:

intersection = my_set1 & my_set2

Разность двух множеств включает в себя все элементы, которые есть в первом множестве, но которых нет во втором:

diff = my_set1.difference (my_set2)

Или аналог:

diff = my_set1 – my_set2

Симметричная разность двух множеств включает в себя все элементы, которые есть только в одном из этих множеств:

symm_diff = my_set1.symmetric_difference (my_set2)

Или аналогичный вариант:

symm_diff = my_set1 ^ my_set2

Люди часто путают обозначения | и &, поэтому рекомендуется вместо них писать s1.union (s2) и s1.intersection (s2). Операции – и ^ перепутать сложнее, их можно записывать прямо так.

```
s1 = {'a', 'b', 'c'}
```

```
s2 = {'a', 'c', 'd'}
union = s1.union (s2) # {'a', 'b', 'c', 'd'}
intersection = s1.intersection (s2) # {'a', 'c'}
diff = s1 – s2 # {'b'}
symm_diff = s1 ^ s2 # {'b', 'd'}
```

4. Сравнение множеств

Все операторы сравнения множеств, а именно ==, <,>, <=,> =, возвращают True, если сравнение истинно, и False в противном случае.

Важно

Множества считаются равными, если они содержат одинаковые наборы элементов. Равенство множеств, как в случае с числами и строками, обозначается оператором ==.

Неравенство множеств обозначается оператором!=. Он работает противоположно оператору ==.

```
if set1 == set2:
print («Множества равны»)
else:
print («Множества не равны»)
```

Обратите внимание на то, что у двух равных множеств могут быть разные порядки обхода, например, из-за того, что элементы в каждое из них добавлялись в разном порядке.

Теперь перейдём к операторам <=,> =. Они означают «является подмножеством» и «является надмножеством».

Подмножество и надмножество

Подмножество – это некоторая выборка элементов множества, которая может быть как меньше множества, так и совпадать с ним, на что указывают символы « <» и «=» в операторе <=. Наоборот, надмножество включает все элементы некоторого множества и, возможно, какие-то ещё.

```
s1 = {'a', 'b', 'c'}
print (s1 <= s1) # True

s2 = {'a', 'b'}
print (s2 <= s1) # True
s3 = {'a'}
print (s3 <= s1) # True
s4 = {'a', 'z'}
print (s4 <= s1) # False
```

Операция s1 <s2 означает «s1 является подмножеством s2, но целиком не совпадает с ним». Операция s1> s2 означает «s1 является надмножеством s2, но целиком не совпадает с ним».

Задачи
Таблица умножения
Выведите таблицу умножения n*n. Лестница

Помогите Владу написать программу, принимающую на вход целое положительное число N, и выводящую на экран последовательность от 1 до N «по ступенькам».
1
2 3
…

Простые числа

Напишите программу, которая выводит все простые числа, меньшие данного натурального числа.

Строки. Индексация

Аннотация

На этом занятии мы углубим свои знания о строках. Теперь мы сможем не только считывать строку, но и работать с ней, в том числе делать посимвольный перебор.

1. Строка как коллекция

На прошлом занятии мы познакомились с коллекцией, которая называется **множество**. Вспомним, что основная особенность коллекций – возможность хранить несколько значений под одним именем. Можно сказать, что коллекция является **контейнером** для этих значений.

Но ещё до изучения множеств мы уже знали тип данных, который ведёт себя подобно коллекции. Этот тип данных – строка. Действительно, ведь строка фактически является последовательностью символов. В некоторых языках программирования есть специальный тип данных **char**, позволяющий хранить один символ. В Python такого типа данных нет, поэтому можно сказать, что строка – это **последовательность односимвольных строк**.

2. Что мы знаем о строках

Давайте вспомним, что мы уже знаем о работе со строками в языке Python. Мы умеем создавать строки четырьмя способами: задавать напрямую, считывать с клавиатуры функцией input (), преобразовывать число в строку функцией str и склеивать из двух других строк операцией +. Кроме того, мы умеем узнавать длину строки, используя функцию len, и проверять, является ли одна строка частью другой, используя операцию in:

```
fixed_word = ’опять»
print (fixed_word)
word = input ()
print (word)
number = 25
number_string = str (number)
print (number_string)
word_plus_number = fixed_word + number_string
print (word_plus_number)
print (len (word_plus_number))
print (’оп» in word_plus_number)
```

3. Индексация в строках

В отличие от множеств, в строках важен порядок элементов (символов). Действительно, если множества {1, 2, 3} и {3, 2, 1} – это одинаковые множества, то строки «МИР» и «РИМ» – это две совершенно разные строки. Наличие порядка даёт нам возможность пронумеровать символы. Нумерация символов начинается с 0:

Важно

По индексу можно получить соответствующий ему символ строки. Для этого нужно после самой строки написать в квадратных скобках индекс символа.

```
word = «привет»
initial_letter = word [0]
print (initial_letter) # сделает то же, что print («п»)
other_letter = word [3]
print (other_letter) # сделает то же, что print («в»)
```

Естественно, в этом примере word с тем же успехом можно было считать с клавиатуры через input (). Тогда мы не могли бы заранее сказать, чему равны переменные initial_letter и other_letter.

А что будет, если попытаться получить букву, номер которой слишком велик? В этом случае Python выдаст ошибку:

```
word = «привет»
print (word [6]) # builtins.IndexError: string index out of range
```

Конечно, номер в квадратных скобках – не обязательно фиксированное число, которое прописано в самой программе. Его тоже можно считать с клавиатуры или получить в результате арифметического действия.

```
word = «привет»
number_of_letter = int (input ()) # предположим, пользователь ввёл 3
print (word [number_of_letter]) # тогда будет выведена буква «в»
```

Важно

Кроме «прямой» индексации (начинающейся с 0), в Python разрешены отрицательные индексы: word [-1] означает последний символ строки word, word [-2] – предпоследний, и так далее.

А можно ли, используя индексацию, изменить какой-либо символ строки? Давайте проверим:

```
word = «карова’ # Написали слово с ошибкой
word [1] = ’o’ # Пробуем исправить, но:
# TypeError: ’str’ object does not support item assignment
```

Важно

Интерпретатор Python выдаёт ошибку – значит, изменить отдельный символ строки невозможно, т.е. строка относится к неизменяемым типам данных в Python.

4. Перебор элементов строки

В предыдущем уроке мы узнали, что цикл for можно использовать для перебора элементов множества. Таким же образом можно использовать цикл for, чтобы перебрать все буквы в слове:

```
text = ’hello, my dear friends!»
vowels = 0
for letter in text:
if letter in {’a’, ’e’, ’i’, ’o’, ’u’, ’y’}:
vowels += 1
```

print (vowels)

Но, так как символы в строке пронумерованы, у нас есть ещё один способ перебрать все элементы в строке – перебрать все индексы, используя уже знакомую нам конструкцию for i in range (…).

```
text = ’hello, my dear friends!»
vowels = 0
for i in range (len (text)):
if text [i] in ’aeiouy’:
vowels += 1
print (vowels)
```

5. Хранение текстов в памяти компьютера

Давайте немного поговорим о том, как строки хранятся в памяти компьютера.

Важно

Поскольку компьютер «умеет» хранить только двоичные числа, для записи нечисловой информации (текстов, изображений, видео, документов) прибегают к **кодированию**.

Самый простой случай кодирования – сопоставление кодов текстовым символам.

Один самых распространенных форматов такого кодирования – таблица ASCII (American standard code for information interchange).

32 -	48 - 0	64 - @	80 - P	96 - '	112 - p
33 - !	49 - 1	65 - A	81 - Q	97 - a	113 - q
34 - "	50 - 2	66 - B	82 - R	98 - b	114 - r
35 - #	51- 3	67 - C	83 - S	99 - c	115 - s
36 - $	52 - 4	68 - D	84 - T	100 - d	116 - t
37 - %	53 - 5	69 - E	85 - U	101 - e	117 - u
38 -&	54 - 6	70 - F	86 - V	102 - f	118 - v
39 - '	55 - 7	71 - G	87 - W	103 - g	119 - w
40 - (	56 - 8	72 - H	88 - X	104 - h	120 - x
41 -)	57 - 9	73 - I	89 - Y	105 - i	121 - y
42 - *	58 - :	74 - J	90 - Z	106 - j	122 - z
43 - +	59 - ;	75 - K	91 - [	107 - k	123 - {
44 - ,	60 - <	76 - L	92 - \	108 - l	124 - |
45 - -	61 - =	77 - M	93 - j	109 - m	125 - }
46 - .	62 ->	78 - N	94 - ^	110 - n	126 - ~
47 - /	63 - ?	79 - O	95 - ÷	111 - o	127 - ⌂
48 -0	64 - @	80 - P	96 -	112 - p	

Изначально в этой таблице каждому символу был поставлен в соответствие 7-битный код, что позволяло идентифицировать 128 различных символов. В таблице вы не видите символы с кодами, меньшими 32, так как они являются служебными и не предназначены для непосредственного вывода на экран (пробел, перевод строки, табуляция и т.д.).

Этого хватало на латинские буквы обоих регистров, знаки препинания и спецсимволы – например перевод строки или разрыв страницы. Позже код расширили до 1 байта, что позволяло хранить уже 256 различных значений: в таблицу помещались буквы второго алфавита (например, кириллица) и дополнительные графические элементы (псевдографика).

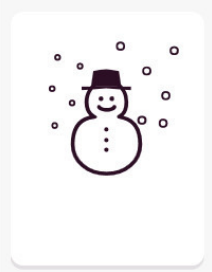
В некоторых относительно низкоуровневых языках (например в C) можно в любой момент перейти от представления строки в памяти к последовательности байтов, начинающейся по какому-либо адресу.

Сейчас однобайтные кодировки отошли на второй план, уступив место Юникоду.

Юникод

Юникод – это таблица, которая содержит соответствия между числом и каким-либо знаком, причем количество знаков может быть любым. Это позволяет одновременно использовать любые символы любых алфавитов, а также дополнительные графические элементы. Кроме того, в Юникоде каждый символ помимо кода имеет некоторые свойства: например, буква это или цифра. Это позволяет более гибко работать с текстами.

В Юникод все время добавляются новые элементы, а сам размер этой таблицы не ограничен и будет только расти, поэтому сейчас при хранении в памяти одного юникод-символа может потребоваться от 1 до 8 байт. Отсутствие ограничений привело к тому, что стали появляться символы на все случаи жизни. Например, есть несколько снеговиков.



Этого вы можете увидеть, если наберете:
print (»**\u**2603»)

В консоли увидим снеговика:
>>> ☃

Важно понять, что все строки в Python хранятся именно как последовательность юникод-символов.

Важно

Для того, чтобы узнать код некоторого символа, существует функция **ord** (*от order – порядок*).

ord («Б»)
>>> 1041

Важно

Зная код, всегда можно получить соответствующий ему символ.
Для этого существует функция **chr** (*от character – символ*):

chr (1041)

>>>«Б»

Функции ord и chr часто работают в паре. Попробуйте, например, предположить, что будет выведено на экран в результате работы следующей программы:

for i **in** range (26):

print (chr (ord («А») + i))

Задачи
Вновь квадраты

Выведите все квадраты чисел от 1 до n при помощи лишь одной строчки кода.

Каждый охотник желает знать…

Пользователь вводит n, если n делиться на семь, то выведите все цвета радуги по порядку, если нет, то начиная с «Красного» отсчитывайте n-ый элемент (если n> 7, то идите по кругу), затем 2n, 3n … 7n.

Строки. Срезы

Аннотация

На этом занятии мы продолжим отрабатывать навыки работы со строкой. А также познакомимся с новым методом извлечения подстроки – срезами.

1. Работа со строками (повторение)

Рассмотрим еще одну задачу. Билет называют счастливым по-питерски, если сумма цифр его номера, стоящих на чётных местах, равна сумме цифр, стоящих на нечётных местах. Нам необходимо написать программу, которая определяет является ли билет счастливым по-питерски.

Если рассматривать номер билета как строку, состоящую из цифр, то задача сводится к подсчёту суммы цифр, стоящих на позициях 0, 2, 4,… и суммы цифр, стоящих на позициях 1, 3, 5,… Чтобы перебрать элементы, мы можем воспользоваться конструкцией for i in range (…), указав шаг 2. Тогда соответствующий фрагмент программы может выглядеть следующим образом:

```
number = input ()
odd = even = 0
for i in range (0, len (number), 2):
odd += int (number [i])
for i in range (1, len (number), 2):
even += int (number [i])
if odd == even:
print («Счастливый по-питерски!»)
```

Подумайте, как можно решить данную задачу за один цикл.

2. Срезы строк

На примере разобранной задачи мы увидели, что перебор элементов строки с помощью конструкции for i in range (…) является достаточно гибким: можно перебрать не все индексы, можно идти с шагом, скажем, 2 или даже —1, то есть в обратном порядке. Но существует способ без всякого цикла преобразовать строку нужным образом: взять отдельный её кусок, символы с нечетными номерами и т. д. Этот способ – **срез (slice)**.

Срез строки

В самом простом варианте **срез строки** – это её кусок от одного индекса включительно и до другого – не включительно (как для range). То есть это новая, более короткая строка.

Срез записывается с помощью квадратных скобок, в которых указывается начальный и конечный индекс, разделённые двоеточием.

```
text = «Hello, world!»
print (text [0:5])
print (text [7:12])
```

Если не указан **начальный индекс**, срез берётся от начала (от 0). Если не указан **конечный индекс**, срез берётся до конца строки. Попробуйте предположить, что будет выведено на экран, если в предыдущей программе записать срезы следующим образом:

```
text = «Hello, world!»
print (text [:5])
print (text [7: ])
```

Разрешены отрицательные индексы для отсчёта с конца списка. В следующем примере из строки, содержащей фамилию, имя и отчество, будет извлекаться фамилия.

```
full_name = «Иванов И. И.»
surname = full_name [:-6]
```

Как и для range, в параметры среза можно добавить третье число – **шаг обхода**. Этот параметр не является обязательным и записывается через второе двоеточие. Вот как может выглядеть программа «счастливый билет», если решать её с помощью срезов:

```
number = input ()
odd = even = 0
# срез будет от начала строки до конца с шагом два: 0, 2, 4,…
for n in number [::2]:
odd += int (n)
# срез от второго элемента строки до конца с шагом два: 1, 3, 5,
…
for n in number [1::2]:
even += int (n)
if odd == even:
print («Счастливый по-питерски!»)
```

Шаг может быть и отрицательным – для прохода по строке в обратном порядке. Если в этом случае не указать начальный и конечный индекс среза, ими станут последний и первый индексы строки, соответственно (а не наоборот, как при положительном шаге):

```
text = «СЕЛ В ОЗЕРЕ БЕРЕЗОВ ЛЕС»
text_reversed = text [::-1]
print (text == text_reversed)
```

Итак, с помощью квадратных скобок можно получить доступ как к одному символу строки, так и к некоторой последовательности символов (причём совсем не обязательно идущих подряд!).

Задачи

Города – 1

Пользователь (или несколько пользователей за одним компьютером) вводит слова. Начиная со второго введённого слова, программа проверяет, совпадает ли первая буква свежевведённого слова с последней буквой предыдущего. Если да, то программа работает дальше (считывает очередное слово). Если нет – выводит последнее на этот момент введённое слово и завершает работу.

Повторение -…

Пользователь вводит строку, а программа должна удвоить каждую букву и вывести новую строчку.

Знакомство со списками

Аннотация

В уроке рассматривается новый тип данных – списки (list), обращение к элементам списка по индексу (аналогично строкам, но с возможностью изменения элементов списка) и метод append. А также вопросы перебора элементов списка и срезов списка.

1. Списки

Мы уже знаем тип данных, который называется **множество** и является **коллекцией (контейнером)**, то есть позволяет хранить несколько элементов данных, и тип строка, который тоже обладает свойствами коллекции. Сегодня мы познакомимся с ещё одним типом-коллекцией, который называется **список** (list). Никогда не создавайте переменные с таким именем!

Списки

Списки являются очень гибкой структурой данных и широко используются в программах. Давайте рассмотрим основные свойства списка в сравнении с теми коллекциями, которые мы уже знаем:

– Список хранит несколько элементов под одним именем (*как и множество*)

– Элементы списка могут повторяться (*в отличие от множества*)

– Элементы списка упорядочены и проиндексированы, доступна операция среза (*как в строке*)

– Элементы списка можно изменять (*в отличие от строки*)

– Элементами списка могут быть значения любого типа: целые и действительные числа, строки и даже другие списки

Конец ознакомительного фрагмента.