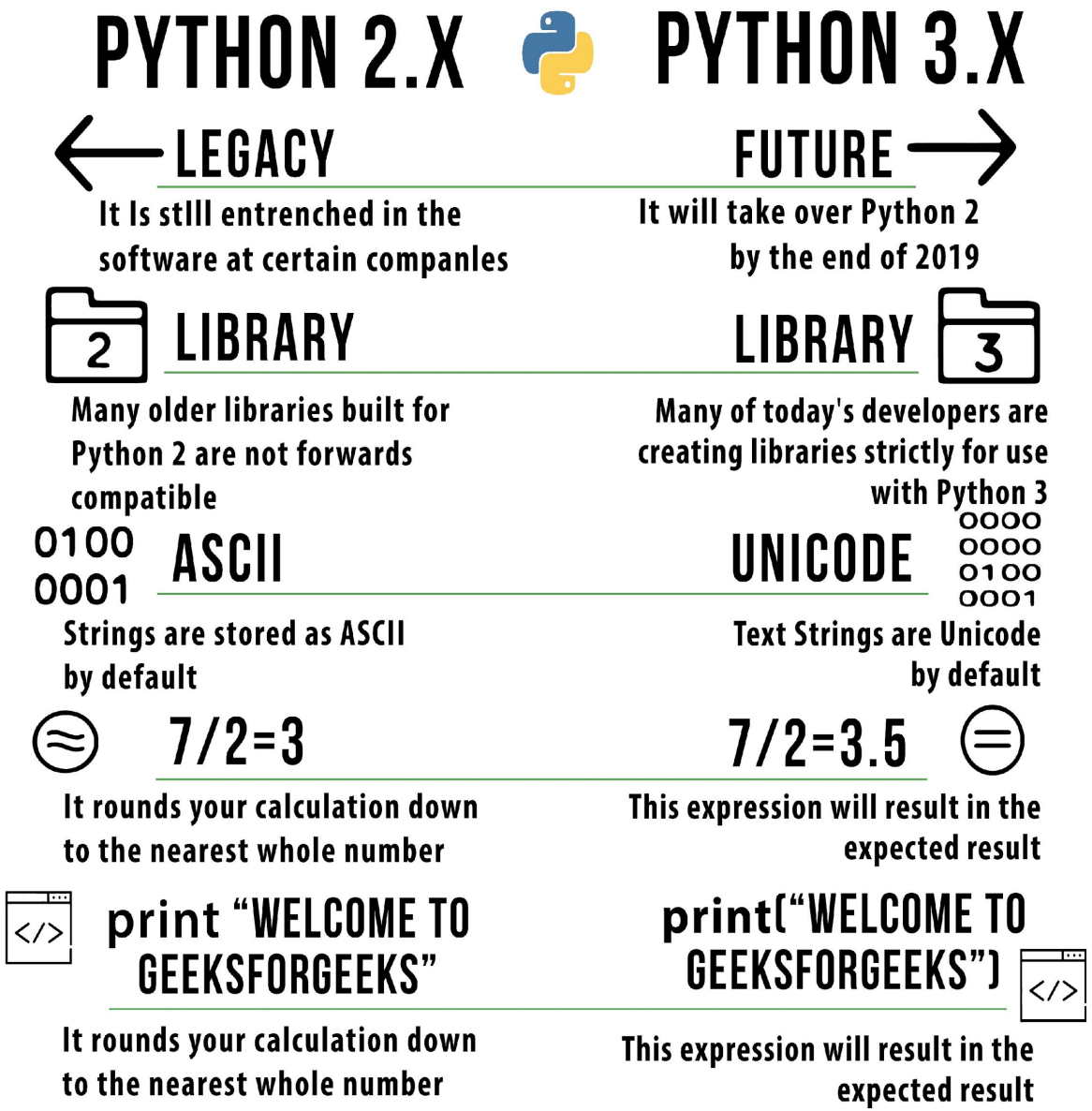
PYTHON КАК УНИВЕРСАЛЬНЫЙ ИНСТРУМЕНТ РЕАЛИЗАЦИИ КОЛИЧЕСТВЕННЫХ ИССЛЕДОВАНИЙ

Борис Михайлович Лямин
к.э.н. старший преподаватель
ВШСиТ СПБПУ

lyamin.bm@gmail.com

ПОЧЕМУ PYTHON?

- Python невероятно эффективен: ваши программы делают больше, чем многие другие языки, в меньшем объеме кода.
- Синтаксис Python позволяет писать «чистый» код. Ваш код будет легко читаться, у вас будет меньше проблем с отладкой и расширением программ по сравнению с другими языками.
- Python используется для разных целей: построения веб-приложений, решений бизнес-задач и разработки внутренних инструментов для всевозможных интересных проектов.
- Развитое сообщество Python.



PYTHON 2 *VS* PYTHON 3

СЕЙЧАС ДОСТУПНЫ ДВЕ ВЕРСИИ PYTHON: PYTHON 2 И БОЛЕЕ НОВАЯ ВЕРСИЯ PYTHON 3.

ВЫПОЛНЕНИЕ ФРАГМЕНТОВ КОДА PYTHON

В поставку Python входит интерпретатор, который выполняется в терминальном окне и позволяет опробовать фрагменты кода Python без сохранения и запуска всей программы. Пример кода:

```
print("Hello Python interpreter!")

Hello Python interpreter!
```

Жирным шрифтом выделен текст, который вы вводите и выполняете нажатием клавиши Enter.

В мире программирования издавна принято начинать освоение нового языка с программы, выводящей на экран сообщение *Hello world!* — считается, что это приносит удачу. На языке Python программа Hello World состоит всего из одной строки:

```
print("Hello world!")
```

PYTHON ONLINE

Существует великое множество online интерпретаторов python. При отсутствии возможности установки, вы можете ими воспользоваться. Вот примеры:

1. https://www.onlinegdb.com/online_python_compiler
2. https://repl.it/languages/python3
3. https://rextester.com/l/python3_online_compiler

ЧАСТЬ I

ПЕРЕМЕННЫЕ И ПРОСТЫЕ ТИПЫ ДАННЫХ

ПЕРЕМЕННЫЕ

Попробуем использовать переменную в программе hello_world .py. Добавьте новую строку в начало файла и измените вторую строку:

```
message = "Hello Python world!»
print(message)
```

Запустите программу и посмотрите, что получится. Программа выводит уже знакомый результат:

```
Hello Python world!
```

В программу добавилась переменная с именем message. В каждой переменной хранится *значение*, то есть данные, связанные с переменной. В данном случае значением является текст “Hello Python world!”.

Добавление переменной немного усложняет задачу интерпретатора Python. Во время обработки первой строки он связывает текст “Hello Python world!” с переменной message. А когда интерпретатор доберется до второй строки, он выводит на экран значение, связанное с именем message.

ПЕРЕМЕННЫЕ

Давайте немного расширим эту программу hello_world .py, чтобы она выводила второе сообщение. Добавьте в hello_world .py пустую строку, а после нее еще две строки кода:

```
message = "Hello Python world!»
print(message)
message = "Hello Python Crash Course world!"
print(message)
```

Теперь при выполнении hello_world .py на экране должны появляться две строки:

```
Hello Python world!
Hello Python Crash Course world!
```

Вы можете в любой момент изменить значение переменной в своей программе. Python всегда отслеживает его текущее состояние.

ПРАВИЛА И РЕКОМЕНДАЦИИ ПРИ ВЫБОРЕ ИМЕН И ИСПОЛЬЗОВАНИЕ ПЕРЕМЕННЫХ

При работе с переменными в языке Python необходимо соблюдать некоторые правила и рекомендации. Нарушение правил приведет к ошибке; рекомендации всего лишь помогают писать более понятный и удобочитаемый код. Работая с переменными, помните о следующем.

- Имена переменных могут состоять только из букв, цифр и символов подчеркивания. Они могут начинаться с буквы или символа подчеркивания, но не с цифры. Например, переменной можно присвоить имя message_1, но не 1_message.
- Пробелы в именах переменных запрещены, а для разделения слов в именах переменных используются символы подчеркивания. Например, имя greeting_ message допустимо, а имя greeting message вызовет ошибку.
- Не используйте имена функций и ключевые слова Python в качестве имен переменных; иначе говоря, не используйте слова, которые зарезервированы в Python для конкретной цели, например слово print.
- Имена переменных должны быть короткими, но содержательными. Например, имя name лучше n, имя student_name лучше s_n, а имя name_length лучше length_of_persons_name.
- Будьте внимательны при использовании строчной буквы l и прописной буквы O, потому что они похожи на цифры 1 и 0.

ПРЕДОТВРАЩЕНИЕ ОШИБОК В ИМЕНАХ ПРИ ИСПОЛЬЗОВАНИИ ПЕРЕМЕННЫХ

Для начала напишем код с намеренно внесенной ошибкой. Введите следующий фрагмент (неправильно написанное слово *mesage* выделено жирным шрифтом):

```
message = "Hello Python Crash Course reader!"
print(mesage)
```

Когда в программе происходит ошибка, интерпретатор Python всеми силами старается помочь вам в поиске причины. Если программа не выполняется нормально, интерпретатор предоставляет данные *трассировки* — информацию о том, в каком месте кода находился интерпретатор при возникновении проблем. Ниже приведен пример трассировки, которую выдает Python после случайной опечатки в имени переменной:

```
Traceback (most recent call last):
❶ File "hello_world.py", line 2, in <module> ❷
print(mesage)
❸ NameError: name 'mesage' is not defined
```

Строка 1 сообщает, что ошибка произошла в строке 2 файла hello_world .py. Интерпретатор выводит номер строки, чтобы вам было проще найти ошибку 2, и сообщает тип обнаруженной ошибки 3. В данном случае была обнаружена ошибка в имени: переменная с указанным именем (mesage) не определена. Другими словами, Python не распознает имя переменной. Обычно такие ошибки возникают в том случае, если вы забыли присвоить значение переменной перед ее использованием или ошиблись при вводе имени.

СТРОКИ

Строка представляет собой простую последовательность символов. Любая последовательность символов, заключенная в кавычки, в Python считается строкой; при этом строки могут быть заключены как в одиночные, так и в двойные кавычки:

```
"This is a string.»
'This is also a string.'
```

Это правило позволяет использовать внутренние кавычки и апострофы в строках:

```
'I told my friend, "Python is my favorite language!»’
"The language 'Python' is named after Monty Python, not the snake."
"One of Python's strengths is its diverse and supportive community."
```

ТИПИЧНЫЕ ОПЕРАЦИИ СО СТРОКАМИ: ИЗМЕНЕНИЕ РЕГИСТРА СИМВОЛОВ В СТРОКАХ

```
name = "ada lovelace"
print(name.title())
```

```
Ada Lovelace
```

В этом примере в переменной **name** сохраняется строка, состоящая из букв нижнего регистра "ada lovelace". За именем переменной в команде print() следует вызов метода **title()**. *Метод* представляет собой действие, которое Python выполняет с данными. Точка (.) после name в конструкции name.title() приказывает Python применить метод title() к переменной name. За именем метода всегда следует пара круглых скобок, потому что методам для выполнения их работы часто требуется дополнительная информация. Эта информация указывается в скобках. Функции title() дополнительная информация не нужна, поэтому в круглых скобках ничего нет.

Метод title() преобразует первый символ каждого слова в строке к верхнему регистру, тогда как все остальные символы выводятся в нижнем регистре. Например, данная возможность может быть полезна, если в вашей программе входные значения Ada, ADA и ada должны рассматриваться как одно и то же имя, и все они должны отображаться в виде Ada.

ТИПИЧНЫЕ ОПЕРАЦИИ СО СТРОКАМИ: ИЗМЕНЕНИЕ РЕГИСТРА СИМВОЛОВ В СТРОКАХ

Для работы с регистром также существуют другие полезные методы. Например, все символы строки можно преобразовать к верхнему или нижнему регистру:

```
name = "Ada Lovelace "
print(name.upper())
print(name.lower())
```

```
ADA LOVELACE
ada lovelace
```

Метод lower() особенно полезен для хранения данных. Нередко программист не может рассчитывать на то, что пользователи введут все данные с точным соблюдением регистра, поэтому строки перед сохранением преобразуются к нижнему регистру. Затем, когда потребуется вывести информацию, используется регистр, наиболее подходящий для каждой строки.

ТИПИЧНЫЕ ОПЕРАЦИИ СО СТРОКАМИ: КОНКАТЕНАЦИЯ

Также часто возникает необходимость в объединении строк. Представьте, что имя и фамилия хранятся в разных переменных и вы хотите объединить их для вывода полного имени:

```
first_name = "ada"
last_name = "lovelace"
full_name = first_name + " " + last_name
print(full_name)
```

```
ada lovelace
```

Для объединения строк в Python используется знак «плюс» (+). В приведенном примере полное имя строится объединением first_name, пробел и last_name. Такой способ объединения строк называется **конкатенацией**. Вы можете использовать конкатенацию для построения сложных сообщений с информацией, хранящейся в переменных.

ТИПИЧНЫЕ ОПЕРАЦИИ СО СТРОКАМИ: КОНКАТЕНАЦИЯ

Рассмотрим пример:

```
first_name = "ada"
last_name = "lovelace"
full_name = first_name + " " + last_name
print("Hello, " + full_name.title() + "!")
```

```
Hello, Ada Lovelace!
```

Полное имя используется для вывода приветственного сообщения, а метод title() обеспечивает правильное форматирование имени. Этот фрагмент возвращает простое, хорошо отформатированное сообщение.

ТИПИЧНЫЕ ОПЕРАЦИИ СО СТРОКАМИ: КОНКАТЕНАЦИЯ

Конкатенацией также можно воспользоваться для построения сообщения, которое затем сохраняется в переменной:

```
first_name = "ada"
last_name = "lovelace"
full_name = first_name + " " + last_name
message = "Hello, " + full_name.title() + "!"
print(message)
```

```
Hello, Ada Lovelace!
```

Этот код также выводит сообщение “Hello, Ada Lovelace!”, но сохранение текста сообщения в переменной существенно упрощает завершающую команду печати.

ТИПИЧНЫЕ ОПЕРАЦИИ СО СТРОКАМИ: ТАБУЛЯЦИИ И РАЗРЫВЫ СТРОК

В программировании термином «**пропуск**» (whitespace) называются такие непечатаемые символы, как пробелы, табуляции и символы конца строки. Пропуски структурируют текст, чтобы пользователю было удобнее читать его. Для включения в текст позиции табуляции используется комбинация символов **\t**:

```
>>> print("Python")
Python

>>> print("\tPython")
    Python
```

Разрывы строк добавляются с помощью комбинации символов \n:

```
>>> print("Languages:\nPython\nC\nJavaScript")
Languages:
Python
C
JavaScript
```

ТИПИЧНЫЕ ОПЕРАЦИИ СО СТРОКАМИ: ТАБУЛЯЦИИ И РАЗРЫВЫ СТРОК

Табуляции и разрывы строк могут сочетаться в тексте. Скажем, последовательность **"\n\t"** приказывает Python начать текст с новой строки, в начале которой располагается табуляция. Следующий пример демонстрирует вывод одного сообщения с разбиением на четыре строки:

```
>>> print("Languages:\n\tPython\n\tC\n\tJavaScript")
Languages:
    Python
    C
    JavaScript
```

ЗАДАНИЕ 1

1. **Личное сообщение**: сохраните имя пользователя в переменной и выведите сообщение, предназначенное для конкретного человека. Сообщение должно быть простым, например: “Hello Eric, would you like to learn some Python today?”
2. **Регистр символов в именах**: сохраните имя пользователя в переменной и выведите его в нижнем регистре, в верхнем регистре и с капитализацией начальных букв каждого слова .
3. **Знаменитая цитата**: найдите известное высказывание, которое вам понравилось . Выведите текст цитаты с именем автора . Результат должен выглядеть примерно так (включая кавычки):

   Albert Einstein once said, "A person who never made a

   mistake never tried anything new.»
4. **Знаменитая цитата 2**: повторите упражнение 3, но на этот раз сохраните имя автора цитаты в переменной famous_person . Затем составьте сообщение и сохраните его в новой переменной с именем message. Выведите свое сообщение.

ЗАДАНИЕ 2

Используя переменные создайте предложение состоящее из не менее 10 слов и содержащее прямую речь. При этом текст в переменных должен быть в нижнем регистре без знаков препинания и пробелов.

При помощи полученных на прошлом занятии знаний и команды **print** вывести стилистически и пунктуационно грамотное предложение, обязательно изменяя регистр слов, добавляя пробелы, знаки препинания, переносы и табуляцию.

ЦЕЛЫЕ ЧИСЛА

В Python с целыми числами можно выполнять операции сложения (+), вычитания (-), умножения (*) и деления (/).

```
>>> 2 + 3
5
>>> 3 - 2
1
>>> 2 * 3
6
>>> 3 / 2
1.5
```

Для представления операции возведения в степень в Python используется сдвоенный знак умножения:

```
>>> 3 ** 2
9
>>> 3 ** 3
27
>>> 10 ** 6
1000000
```

ЦЕЛЫЕ ЧИСЛА

В Python также существует определенный порядок операций, что позволяет использовать несколько операций в одном выражении. Круглые скобки используются для изменения порядка операций, чтобы выражение могло вычисляться в нужном порядке. Пример:

```
>>> 2 + 3 * 4
14
>>> (2 + 3) * 4
20
```

Пробелы в этих примерах не влияют на то, как Python вычисляет выражения; они просто помогают быстрее найти приоритетные операции при чтении кода.

ВЕЩЕСТВЕННЫЕ ЧИСЛА

В Python числа, имеющие дробную часть, называются **вещественными** (или «числами с плавающей точкой»). Обычно разработчик может просто пользоваться дробными значениями, не особенно задумываясь об их поведении. Просто введите нужные числа, а Python скорее всего сделает именно то, что вы от него хотите:

```
>>> 0.1 + 0.1
0.2
>>> 0.2 + 0.2
0.4
>>> 2 * 0.1
0.2
>>> 2 * 0.2
0.4
```

```
>>> 7 / 3
2.3333333333335
>>> 7 // 3
2
>>> 7 ** 0,5
2,645751....
```

```
>>> a = 10

>>> import math

math.sqrt(a)

3,162277.....
```

ПРЕДОТВРАЩЕНИЕ ОШИБОК ТИПОВ С ИСПОЛЬЗОВАНИЕМ ФУНКЦИИ STR()

Часто значение переменной должно использоваться внутри сообщения. Допустим, вы хотите поздравить пользователя с днем рождения. И вы написали для этого следующий код:

```
age = 23
message = "Happy " + age + "rd Birthday!»
print(message)
```

Казалось бы, программа должна вывести простое приветствие: *Happy 23rd birthday!* Но, если запустить ее, появляется сообщение об ошибке:

```
Traceback (most recent call last):
    File "birthday.py", line 2, in <module>          message =
"Happy " + age + "rd Birthday!"
❶ TypeError: Can't convert 'int' object to str implicitly
```

ПРЕДОТВРАЩЕНИЕ ОШИБОК ТИПОВ С ИСПОЛЬЗОВАНИЕМ ФУНКЦИИ STR()

На этот раз произошла **ошибка типа**. Это означает, что Python не понимает, какую информацию вы используете. В данном примере Python видит, что в точке 1 используется переменная с целочисленным значением **(int)**, но не знает, как следует интерпретировать это значение. Дело в том, что переменная может представлять как число 23, так и пару отдельных символов 2 и 3. При таком использовании целых чисел в строках необходимо явно указать, что целое число должно использоваться как строка из символов. Для этого переменная передается функции **str()**, преобразующей не строковые значения к строковому виду:

```
age = 23
message = "Happy " + str(age) + "rd Birthday!"
print(message)
```

```
Happy 23rd Birthday!
```

Теперь Python понимает, что вы хотите преобразовать числовое значение 23 в строку и вывести символы 2 и 3 в составе поздравления. Ожидаемый результат выводится без всяких ошибок.

ЗАДАНИЕ 3

1. Используя переменные сохраните 3 целых числа и 2 вещественных числа. **Выведите на экран результат суммы, вычитания, умножения, деления и корня из этих чисел в любом порядке**
2. **Попробуйте найти ответ данного выражения**

$$\frac{75}{15 + 5 * (-3)}$$

3. **Найдите результат вычисления чисел**

2014**14

2014,0**14

4. **Выведите числа в виде десятичной дроби**

1,048576e+06

6,0221456e-03

КОММЕНТАРИИ

Комментарии чрезвычайно полезны в любом языке программирования. До сих пор ваши программы состояли только из кода Python. По мере роста объема и сложности кода в программы следует добавлять комментарии, описывающие общий подход к решаемой задаче, — своего рода заметки, написанные на понятном языке.

КАК СОЗДАЮТСЯ КОММЕНТАРИИ?

В языке Python признаком комментария является символ «решетка» (#). Интерпретатор Python игнорирует все символы, следующие в коде после # до конца строки. Пример:

```
# Say hello to everyone.
print("Hello Python people!")
```

```
Hello Python people!
```

Python игнорирует первую строку и выполняет вторую.

ЗАДАНИЕ 4

При помощи комментариев подробно опишите процесс написания кода для математических операций с числами с использованием переменных. Данные для расчета представлены ниже.

1. 1,64 ; 1,95; 1,76; 1,82; 1,92.
2. 18, 29, 37, 46, 54

ЧАСТЬ II

СПИСКИ

ЧТО ТАКОЕ СПИСОК?

Список — это набор элементов, следующих в определенном порядке. Вы можете создать список для хранения букв алфавита, цифр от 0 до 9 или имен всех членов вашей семьи. В список можно поместить любую информацию, причем данные в списке даже не обязаны быть как-то связаны друг с другом. Так как список обычно содержит более одного элемента, рекомендуется присваивать спискам имена во множественном числе: letters, digits, names и т. д.

В языке Python список обозначается квадратными скобками ([]), а отдельные элементы списка разделяются запятыми. Простой пример списка с названиями моделей велосипедов:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles)
```

Если вы прикажете Python вывести список, то на экране появится перечисление элементов списка в квадратных скобках:

```
['trek', 'cannondale', 'redline', 'specialized']
```

ОБРАЩЕНИЕ К ЭЛЕМЕНТАМ СПИСКА

Списки представляют собой упорядоченные наборы данных, поэтому для обращения к любому элементу списка следует сообщить Python позицию (*индекс*) нужного элемента. Чтобы обратиться к элементу в списке, укажите имя списка, за которым следует индекс элемента в квадратных скобках.

Например, название первого велосипеда в списке bicycles выводится следующим образом:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
❶ print(bicycles[0])
```

Синтаксис обращения к элементу показан в точке ❶. Когда мы запрашиваем один элемент из списка, Python возвращает только этот элемент без квадратных скобок или кавычек:

```
trek
```

Именно такой результат должны увидеть пользователи — чистый, аккуратно отформатированный вывод.

ОБРАЩЕНИЕ К ЭЛЕМЕНТАМ СПИСКА

Также можно использовать строковые методы из главы 2 с любым элементом списка. Например, элемент 'trek' можно более аккуратно отформатировать при помощи метода title():

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[0].title())
```

Этот пример выдает такой же результат, как и предыдущий, только название 'Trek' выводится с прописной буквы.

ИНДЕКСЫ НАЧИНАЮТСЯ С НУЛЯ, А НЕ С ЕДИНИЦЫ

Python считает, что первый элемент списка находится в позиции 0, а не в позиции 1. Этот принцип встречается в большинстве языков программирования и объясняется особенностями низкоуровневой реализации операций со списками. Если вы получаете неожиданные результаты, определите, не допустили ли вы простую ошибку «смещения на 1». Второму элементу списка соответствует индекс 1. В этой простой схеме индекс любого элемента вычисляется уменьшением на 1 его позиции в списке. Например, чтобы обратиться к четвертому элементу списка, следует запросить элемент с индексом 3.

В примере выводятся названия велосипедов с индексами 1 и 3:

```
bicycles = ['trek', 'cannondale', 'redline',
'specialized'] print(bicycles[1])
print(bicycles[3])
```

При этом выводится второй и четвертый элементы списка:

```
cannondale
specialized
```

ОБРАЩЕНИЕ К ПОСЛЕДНЕМУ ЭЛЕМЕНТУ СПИСКА

В Python также существует специальный синтаксис для обращения к последнему элементу списка. Если запросить элемент с индексом –1, Python всегда возвращает последний элемент в списке:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[-1])
```

Фрагмент вернет значение 'specialized'. Этот синтаксис весьма полезен, потому что при работе со списками часто требуется обратиться к последним элементам, не зная точное количество элементов в списке. Синтаксис также распространяется на другие отрицательные значения индексов. Индекс –2 возвращает второй элемент от конца списка, индекс –3 — третий элемент от конца и т. д.

ИСПОЛЬЗОВАНИЕ ОТДЕЛЬНЫХ ЭЛЕМЕНТОВ ИЗ СПИСКА

Отдельные значения из списка используются так же, как и любые другие переменные. Например, вы можете воспользоваться конкатенацией для построения сообщения, содержащего значение из списка.

Попробуем извлечь название первого велосипеда из списка и составить сообщение, включающее это значение.

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized’]
❶ message = "My first bicycle was a " + bicycles[0].title() + "."
print(message)
```

В точке ❶ программа строит сообщение, содержащее значение из bicycles[0], и сохраняет его в переменной message. Так создается простое предложение с упоминанием первого велосипеда из списка:

```
My first bicycle was a Trek.
```

ЗАДАНИЕ 3

Попробуйте написать несколько коротких программ, чтобы получить предварительное представление о списках Python.

1. **Имена**: сохраните имена не менее 5 своих друзей в списке с именем names. Выведите имя каждого друга, обратившись к каждому элементу списка (по одному за раз) .
2. **Сообщения**: начните со списка, использованного в упражнении 1, но вместо вывода имени каждого человека выведите сообщение. Основной текст всех сообщений должен быть одинаковым, но каждое сообщение должно включать имя адресата .
3. **Собственный список**: выберите свой любимый вид транспорта (например, мотоциклы или машины) и создайте список с примерами. Используйте свой список для вывода утверждений об элементах типа: «Я хотел бы купить мотоцикл Honda» .

ИЗМЕНЕНИЕ ЭЛЕМЕНТОВ В СПИСКЕ

Синтаксис изменения элемента напоминает синтаксис обращения к элементу списка. Чтобы изменить элемент, укажите имя списка и индекс изменяемого элемента в квадратных скобках; далее задайте новое значение, которое должно быть присвоено элементу.

Допустим, имеется список мотоциклов, и первым элементом списка хранится строка 'honda'. Как изменить значение первого элемента?

motorcycles.py

```
❶ motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)
❷ motorcycles[0] = 'ducati' print(motorcycles)
```

В точке ❶ определяется исходный список, в котором первый элемент содержит строку 'honda'. В точке ❷ значение первого элемента заменяется строкой 'ducati'. Из вывода видно, что первый элемент действительно изменился, а остальные элементы списка сохранили прежние значения:

```
['honda', 'yamaha', 'suzuki']
['ducati', 'yamaha', 'suzuki']
```

ДОБАВЛЕНИЕ ЭЛЕМЕНТОВ В СПИСОК

Новые элементы могут добавляться в списки по разным причинам — например, для появления на экране новых космических кораблей, включения новых данных в визуализацию или добавления новых зарегистрированных пользователей на построенный вами сайт. Python предоставляет несколько способов добавления новых данных в существующие списки.

ПРИСОЕДИНЕНИЕ ЭЛЕМЕНТОВ В КОНЕЦ СПИСКА

Простейший способ добавления новых элементов в список — присоединение элемента в конец списка. Используя список из предыдущего примера, добавим новый элемент 'ducati':

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)

❶ motorcycles.append('ducati’)
print(motorcycles)
```

Метод append () в точке ❶ присоединяет строку 'ducati' в конец списка, другие элементы в списке при этом остаются неизменными:

```
['honda', 'yamaha', 'suzuki’]
['honda', 'yamaha', 'suzuki', 'ducati']
```

ПРИСОЕДИНЕНИЕ ЭЛЕМЕНТОВ В КОНЕЦ СПИСКА

Метод append() упрощает динамическое построение списков. Например, вы можете начать с пустого списка и добавлять в него элементы серией команд append(). В следующем примере в пустой список добавляются элементы 'honda', 'yamaha' и 'suzuki':

```
motorcycles = []
motorcycles.append('honda')
motorcycles.append('yamaha')
motorcycles.append('suzuki’)
print(motorcycles)
```

Полученный список выглядит точно так же, как и списки из предыдущих примеров:

```
['honda', 'yamaha', 'suzuki']
```

ВСТАВКА ЭЛЕМЕНТОВ В СПИСОК

Метод insert() позволяет добавить новый элемент в произвольную позицию списка. Для этого следует указать индекс и значение нового элемента.

```
motorcycles = ['honda', 'yamaha', 'suzuki’]
❶ motorcycles.insert(0, 'ducati')
print(motorcycles)
```

В этом примере в точке ❶ значение 'ducati' вставляется в начало списка. Метод insert() выделяет свободное место в позиции о и сохраняет в нем значение 'ducati'. Все остальные значения списка при этом сдвигаются на одну позицию вправо:

```
['ducati', 'honda', 'yamaha', 'suzuki']
```

УДАЛЕНИЕ ЭЛЕМЕНТОВ ИЗ СПИСКА С ИСПОЛЬЗОВАНИЕМ КОМАНДЫ DEL

Если вам известна позиция элемента, который должен быть удален из списка, воспользуйтесь командой del.

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)
❶ del motorcycles[0]
print(motorcycles)
```

В точке ❶ вызов del удаляет первый элемент, 'honda', из списка motorcycles:

```
['honda', 'yamaha', 'suzuki']
['yamaha', 'suzuki']
```

УДАЛЕНИЕ ЭЛЕМЕНТОВ ИЗ СПИСКА ПО ЗНАЧЕНИЮ

Иногда позиция удаляемого элемента неизвестна. Если вы знаете только значение элемента, используйте метод remove(). Допустим, из списка нужно удалить значение 'ducati':

```
motorcycles = ['honda', 'yamaha', 'suzuki', 'ducati']
print(motorcycles)
❶ motorcycles.remove('ducati’)
print(motorcycles)
```

Код в точке ❶ приказывает Python определить, в какой позиции списка находится значение 'ducati', и удалить этот элемент:

```
['honda', 'yamaha', 'suzuki', 'ducati']
['honda', 'yamaha', 'suzuki']
```

УДАЛЕНИЕ ЭЛЕМЕНТОВ ИЗ СПИСКА ПО ЗНАЧЕНИЮ

Метод remove() также может использоваться для работы со значением, которое удаляется из списка. Следующая программа удаляет значение 'ducati' и выводит причину удаления:

```
❶ motorcycles = ['honda', 'yamaha', 'suzuki', 'ducati']
print(motorcycles)
❷ too_expensive = 'ducati'
❸ motorcycles.remove(too_expensive)
print(motorcycles)
❹ print("\nA " + too_expensive.title() + " is too expensive for me.")
```

После определения списка в точке ❶ значение 'ducati' сохраняется в переменной с именем too_expensive в точке ❷. Затем эта переменная сообщает Python, какое значение должно быть удалено из списка ❸. В точке ❹ значение 'ducati' было удалено из списка, но продолжает храниться в переменной too_expensive, что позволяет вывести сообщение с причиной удаления 'ducati' из списка мотоциклов:

```
['honda', 'yamaha', 'suzuki', 'ducati']
['honda', 'yamaha', 'suzuki']
A Ducati is too expensive for me.
```

УДАЛЕНИЕ ЭЛЕМЕНТОВ ИЗ СПИСКА С ИСПОЛЬЗОВАНИЕМ МЕТОДА POP()

Иногда значение, удаляемое из списка должно как-то использоваться, например, вы хотите получить координаты x и y только что сбитого корабля пришельцев чтобы изобразить взрыв в этой позиции. Соответственно удалить мы его можем при помощи добавления в список неактивных. Метод POP () удаляет последний элемент из списка и позволяет с ним работать после удаления.

```
❶ motorcycles = ['honda', 'yamaha', 'suzuki', 'ducati’]
print(motorcycles)
❷ pop_motorcycles = motorcycles.pop()
❸ print(motorcycles)
❹ print(pop_motorcycles)
```

Сначала в точке ❶ определяется и выводится содержимое списка `motorcycles`. В точке ❷ значение извлекается из списка и сохраняется в переменной с именем `pop_motorcycles`. Вывод измененного списка в точке ❸ показывает, что значение было удалено из списка. Затем выводим извлеченное значение в точке ❹, демонстрируя, что удаленное из списка значение остается доступным в программе.

```
['honda', 'yamaha', 'suzuki', 'ducati']
['honda', 'yamaha', 'suzuki’]
ducati
```

ИЗВЛЕЧЕНИЕ ЭЛЕМЕНТОВ ИЗ ПРОИЗВОЛЬНОЙ ПОЗИЦИИ СПИСКА

Метод POP () может использоваться для удаления элемента в произвольной позиции списка, для этого следует указать индекс удаляемого элемента в круглых скобках.

```
❶ motorcycles = ['honda', 'yamaha', 'suzuki', 'ducati’]
❷ pop_motorcycles = motorcycles.pop(0)
❸ print(‘The first motorcycles I owned was a + pop_motorcycles.title() + ‘.’)
```

Сначала в точке ❶ определяется содержимое списка `motorcycles`. В точке ❷ первый элемент извлекается из списка и сохраняется в переменной с именем `pop_motorcycles`. В точке ❸ выводится сообщение об этом мотоцикле.

```
The first motorcycles I owned was a Honda.
```

ЗАДАНИЕ 4

Следующие упражнения немного сложнее предыдущих, но они предоставляют возможность попрактиковаться в выполнении всех описанных операций со списками .

1. **Список гостей**: если бы вы могли пригласить кого угодно (из живых или умерших) на обед, то кого бы вы пригласили? Создайте список, включающий минимум трех людей, которых вам хотелось бы пригласить на обед. Затем используйте этот список для вывода пригласительного сообщения каждому участнику.
2. **Изменение списка гостей**: вы только что узнали, что один из гостей прийти не сможет, поэтому вам придется разослать новые приглашения. Отсутствующего гостя нужно заменить кем-то другим .
   - Начните с программы из предыдущего упражнения. Добавьте в конец программы команду print для вывода имени гостя, который прийти не сможет.
   - Измените список и замените имя гостя, который прийти не сможет, именем нового приглашенного.
   - Выведите новый набор сообщений с приглашениями – по одному для каждого участника, входящего в список.
3. **Больше гостей**: вы решили купить обеденный стол большего размера. Дополнительные места позволяют пригласить на обед еще трех гостей .
   - Начните с программы из предыдущих упражнений. Добавьте в конец программы команду print, которая выводит сообщение о расширении списка гостей .
   - Добавьте вызов insert() для добавления одного гостя в начало списка.
   - Добавьте вызов insert() для добавления одного гостя в середину списка.
   - Добавьте вызов append() для добавления одного гостя в конец списка.
   - Выведите новый набор сообщений с приглашениями – по одному для каждого участника, входящего в список.

ЗАДАНИЕ 4
(ПРОДОЛЖЕНИЕ)

Следующие упражнения немного сложнее предыдущих, но они предоставляют возможность попрактиковаться в выполнении всех описанных операций со списками .

1. **Сокращение списка гостей**: только что выяснилось, что новый обеденный стол привезти вовремя не успеют, и места хватит только для двух гостей.
   - Начните с программы из предыдущего упражнения. Добавьте команду для вывода сообщения о том, что на обед приглашаются всего два гостя.
   - Используйте метод pop() для последовательного удаления гостей из списка до тех пор, пока в списке не останутся только два человека. Каждый раз, когда из списка удаляется очередное имя, выведите для этого человека сообщение о том, что вы сожалеете об отмене приглашения.
   - Выведите сообщение для каждого из двух человек, остающихся в списке. Сообщение должно подтверждать, что более раннее приглашение остается в силе.
   - Используйте команду del для удаления двух последних имен, чтобы список остался пустым. Выведите список, чтобы убедиться в том, что в конце работы программы список действительно не содержит ни одного элемента.

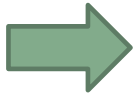
УПОРЯДОЧЕНИЕ СПИСКА

Нередко список создается в непредсказуемом порядке, потому что порядок получения данных от пользователя не всегда находится под вашим контролем. И хотя во многих случаях такое положение дел неизбежно, часто требуется вывести информацию в определенном порядке. В одних случаях требуется сохранить исходный порядок элементов в списке, в других исходный порядок должен быть изменен. Python предоставляет в распоряжение программиста несколько разных способов упорядочения списка в зависимости от ситуации.

ПОСТОЯННАЯ СОРТИРОВКА СПИСКА МЕТОДОМ SORT()

Метод sort() позволяет относительно легко отсортировать список. Предположим, имеется список машин, и вы хотите переупорядочить эти элементы по алфавиту. Чтобы упростить задачу, предположим, что все значения в списке состоят из символов нижнего регистра.

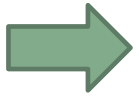
```
cars = ['bmw', 'audi', 'toyota', 'subaru’]
❶ cars.sort()
print(cars)
```



```
['audi', 'bmw', 'subaru', 'toyota']
```

Метод sort() в точке ❶ осуществляет постоянное изменение порядка элементов в списке. Названия машин располагаются в алфавитном порядке, и вернуться к исходному порядку уже не удастся:

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
cars.sort(reverse=True)
print(cars)
```

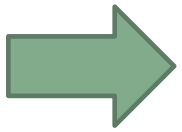


```
['toyota', 'subaru', 'bmw', 'audi']
```

ВРЕМЕННАЯ СОРТИРОВКА СПИСКА ФУНКЦИЕЙ SORTED()

Чтобы сохранить исходный порядок элементов списка, но временно представить их в отсортированном порядке, можно воспользоваться функцией sorted(). Функция sorted() позволяет представить список в определенном порядке, но не изменяет фактического порядка элементов в списке.

```
cars = ['bmw', 'audi', 'toyota', 'subaru’]
❶ print("Here is the original list:")
print(cars)
❷ print("\nHere is the sorted list:")
print(sorted(cars))
❸ print("\nHere is the original list again:")
print(cars)
```



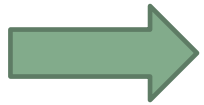
```
Here is the original list:
['bmw', 'audi', 'toyota', 'subaru’]
Here is the sorted list:
['audi', 'bmw', 'subaru', 'toyota']
❶ Here is the original list again:
['bmw', 'audi', 'toyota', 'subaru']
```

Сначала список выводится в исходном порядке ❶, а затем в алфавитном ❷. После того как список будет выведен в новом порядке, в точке ❸, мы убеждаемся в том, что список все еще хранится в исходном порядке.

ВЫВОД СПИСКА В ОБРАТНОМ ПОРЯДКЕ

Чтобы переставить элементы списка в обратном порядке, используйте метод reverse(). Скажем, если список машин первоначально хранился в хронологическом порядке даты приобретения, элементы можно легко переупорядочить в обратном хронологическом порядке:

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
print(cars)
cars.reverse()
print(cars)
```



```
['bmw', 'audi', 'toyota', 'subaru']
['subaru', 'toyota', 'audi', 'bmw']
```

Обратите внимание: метод reverse() не сортирует элементы в обратном алфавит- ном порядке, а просто переходит к обратному порядку списка. Метод reverse() осуществляет постоянное изменение порядка элементов, но вы можете легко вернуться к исходному порядку, снова применив reverse() к обратному списку.

ОПРЕДЕЛЕНИЕ ДЛИНЫ СПИСКА

Вы можете быстро определить длину списка с помощью функции len(). Список в нашем примере состоит из четырех элементов, поэтому его длина равна 4:

```
>>> cars = ['bmw', 'audi', 'toyota', 'subaru']
>>> len(cars)
4
```

Метод len() может пригодиться для определения количества пришельцев, которых необходимо сбить в игре; объема данных, которыми необходимо управлять в визуализации; количества зарегистрированных пользователей на веб-сайте и т. д.

ЗАДАНИЕ 5

1. **Повидать мир**: вспомните хотя бы пять стран, в которых вам хотелось бы побывать.
   - Сохраните названия стран в списке. Проследите за тем, чтобы список не хранился в алфавитном порядке.
   - Выведите список в исходном порядке. Не беспокойтесь об оформлении списка, просто выведите его как обычный список Python.
   - Используйте функцию sorted() для вывода списка в алфавитном порядке без изменения списка .
   - Снова выведите список, чтобы показать, что он по-прежнему хранится в исходном порядке.
   - Используйте функцию sorted() для вывода списка в обратном алфавитном порядке без изменения порядка исходного списка.
   - Снова выведите список, чтобы показать, что исходный порядок не изменился.
   - Измените порядок элементов вызовом reverse(). Выведите список, чтобы показать, что элементы следуют в другом порядке.
   - Измените порядок элементов повторным вызовом reverse(). Выведите список, чтобы показать, что список вернулся к исходному порядку.
   - Отсортируйте список в алфавитном порядке вызовом sort(). Выведите список, чтобы показать, что элементы следуют в другом порядке.
   - Вызовите sort() для перестановки элементов списка в обратном алфавитном порядке. Выведите список, чтобы показать, что порядок элементов изменился.

ЗАДАНИЕ 5
ПРОДОЛЖЕНИЕ

1. **Количество гостей**: в одной из программ из предыдущих упражнений используйте len() для вывода сообщения с количеством людей, приглашенных на обед.
2. **Все функции**: придумайте информацию, которую можно было бы хранить в списке. Например, создайте список гор, рек, стран, городов, языков... словом, чего угодно. Напишите программу, которая создает список элементов, а затем вызывает каждую функцию, упоминавшуюся в этой главе, хотя бы один раз.

ЧАСТЬ III

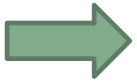
РАБОТА СО СПИСКАМИ

ПЕРЕБОР ВСЕГО СПИСКА

Типичная задача из области программирования — **перебрать все элементы списка и выполнить с каждым элементом одну и ту же операцию**. Например, в компьютерной игре все экранные объекты могут смещаться на одинаковую величину, или в списке чисел к каждому элементу может применяться одна и та же статистическая операция. А может быть, вам нужно вывести все заголовки из списка статей на сайте.. **В ситуациях, требующих применения одного действия к каждому элементу списка, можно воспользоваться циклами *for***

Допустим, имеется список с именами фокусников, и вы хотите вывести каждое имя из списка. Конечно, можно обратиться к каждому элементу по отдельности, но такой подход создает ряд проблем. Во-первых, для очень длинных списков все сведется к однообразным повторениям. Во-вторых, при любом изменении длины списка в программу придется вносить изменения. Цикл for решает обе проблемы: Python будет следить за всеми техническими деталями в своей внутренней реализации.

```
❶ magicians = ['alice', 'david', 'carolina'] ❷
for magician in magicians:
❸  print(magician)
```



```
alice
david
carolina
```

Все начинается с определения списка ❶. В точке ❷ определяется цикл for. Эта строка приказывает Python взять очередное имя из списка и сохранить его в переменной magician. В точке ❸ выводится имя, только что сохраненное в переменной magician. Затем строки ❷ и ❸ повторяются для каждого имени в списке. Этот код можно описать так: «Для каждого фокусника в списке вывести его имя».

ПОДРОБНЕЕ О ЦИКЛАХ

Концепция циклов очень важна, потому что она представляет один из основных способов автоматизации повторяющихся задач компьютером. Например, в простом предыдущем цикле Python сначала читает первую строку цикла:

```
for magician in magicians:
```

Эта строка означает, что нужно взять первое значение из списка magicians и сохранить его в переменной magician. Первое значение в списке — 'alice'. Затем Python читает следующую строку:

```
print(magician)
```

Python выводит текущее значение magician, которое все еще равно 'alice'. Так как в списке еще остались другие значения, Python возвращается к первой строке цикла:

```
for magician in magicians:
```

Python берет следующее значение из списка — 'david' — и сохраняет его в magician. Затем выполняется строка:

```
print(magician)
```

Python снова выводит текущее значение magician; теперь это строка 'david'. Весь цикл повторяется еще раз с последним значением в списке, 'carolina'. Так как других значений в списке не осталось, Python переходит к следующей строке в программе. В данном случае после цикла for ничего нет, поэтому программа просто завершается.

БОЛЕЕ СЛОЖНЫЕ ДЕЙСТВИЯ В ЦИКЛАХ FOR

В цикле for с каждым элементом списка может выполняться практически любое действие. Дополним предыдущий пример, чтобы программа выводила для каждого фокусника отдельное сообщение:

```
magicians = ['alice', 'david', 'carolina’]
for magician in magicians:
❶ print(magician.title() + ", that was a great trick!")
```

Единственное отличие этого кода от предыдущего заключается в том, что в точке ❶ для каждого фокусника строится сообщение с его именем. При первом проходе цикла переменная magician содержит значение 'alice', поэтому Python начинает первое сообщение с имени 'Alice'. При втором проходе сообщение будет начинаться с имени 'David', а при третьем — с имени 'Carolina':

```
Alice, that was a great trick!
David, that was a great trick!
Carolina, that was a great trick!
```

БОЛЕЕ СЛОЖНЫЕ ДЕЙСТВИЯ В ЦИКЛАХ FOR

Тело цикла for может содержать сколько угодно строк кода. Каждая строка с начальным отступом после строки for magician in magicians считается находящейся в цикле и выполняется по одному разу для каждого значения в списке. Таким образом, с каждым значением в списке можно выполнить любые операции на ваше усмотрение.

```
magicians = ['alice', 'david', 'carolina’]
for magician in magicians:
    print(magician.title() + ", that was a great trick!")
❶   print("I can't wait to see your next trick, " + magician. title() + ".\n")
```

Так как обе команды print снабжены отступами, каждая строка будет выполнена по одному разу для каждого фокусника в списке. Символ новой строки ("\n") во второй команде print ❶ вставляет пустую строку после каждого прохода цикла. В результате будет создан набор сообщений, аккуратно сгруппированных для каждого фокусника в списке:

```
Alice, that was a great trick!
I can't wait to see your next trick, Alice.

David, that was a great trick!
I can't wait to see your next trick, David.

Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.
```

ВЫПОЛНЕНИЕ ДЕЙСТВИЙ ПОСЛЕ ЦИКЛА FOR

Что происходит после завершения цикла for? Обычно программа выводит сводную информацию или переходит к другим операциям.

Каждая строка кода после цикла for, не имеющая отступа, выполняется без повторения. Допустим, вы хотите вывести сообщение для всей группы фокусников и поблагодарить их за превосходное представление. Чтобы вывести общее сообщение после всех отдельных сообщений, разместите его после цикла for без отступа:

```
magicians = ['alice', 'david', 'carolina’]
for magician in magicians:
    print(magician.title() + ", that was a great trick!")
    print("I can't wait to see your next trick, " + magician.title() + ".\n")
❶ print("Thank you, everyone. That was a great magic show!")
```

Первые две команды print повторяются по одному разу для каждого фокусника в списке, как было показано ранее. Но поскольку строка ❶ отступа не имеет, это сообщение выводится только один раз:

```
Alice, that was a great trick!
I can't wait to see your next trick, Alice.

David, that was a great trick!
I can't wait to see your next trick, David.

Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.

Thank you, everyone. That was a great magic show!
```

ПРЕДОТВРАЩЕНИЕ ОШИБОК С ОТСТУПАМИ

В Python связь одной строки кода с предшествующей строкой обозначается отступами. В приведенных примерах строки, выводившие сообщения для отдельных фокусников, были частью цикла, потому что они были снабжены отступами. Применение отступов в Python сильно упрощает чтение кода. Фактически отступы заставляют разработчика писать аккуратно отформатированный код с четкой визуальной структурой. В более длинных программах Python могут встречаться блоки кода с отступами нескольких разных уровней. Эти уровни способствуют пониманию общей структуры программы.

Когда разработчики только начинают писать код, работа которого зависит от правильности отступов, в их коде нередко встречаются распространенные ошибки. Например, иногда они расставляют отступы в коде, в котором эти отступы не нужны, или наоборот — забывают ставить отступы в блоках, где это необходимо. Несколько примеров помогут вам избежать подобных ошибок в будущем и успешно исправлять их, когда они встретятся в ваших программах.

ПРОПУЩЕННЫЙ ОТСТУП

Строка после команды for в цикле всегда должна снабжаться отступом. Если вы забудете поставить отступ, Python напомнит вам об этом:

```
    magicians = ['alice', 'david', 'carolina’]
    for magician in magicians:
❶ print(magician)
```

Команда print в точке ❶ должна иметь отступ, но здесь его нет. Когда Python ожидает увидеть блок с отступом, но не находит его, появляется сообщение с указанием номера строки:

```
    line 3
    print(magician)
    ^
    IndentationError: expected an indented block
```

Обычно для устранения подобных ошибок достаточно поставить отступ в строке (или строках), следующей непосредственно после команды for.

ПРОПУЩЕННЫЕ ОТСТУПЫ В ДРУГИХ СТРОКАХ

Иногда цикл выполняется без ошибок, но не выдает ожидаемых результатов. Такое часто происходит, когда вы пытаетесь выполнить несколько операций в цикле, но забываете снабдить отступом некоторые из строк.

Например, вот что происходит, если вы забудете снабдить отступом вторую строку в цикле:

```
magicians = ['alice', 'david', 'carolina’]
for magician in magicians:
    print(magician.title() + ", that was a great trick!")
❶ print("I can't wait to see your next trick, " + magician.title() + ".\n")
```

Команда print в точке ❶ должна быть снабжена отступом, но, поскольку Python находит хотя бы одну строку с отступом после команды for, сообщение об ошибке не выдается. В результате первая команда print будет выполнена для каждого элемента в списке, потому что в ней есть отступ. Вторая команда print отступа не имеет, поэтому она будет выполнена только один раз после завершения цикла. Так как последним значением magician является строка 'carolina', второе сообщение будет выведено только с этим именем:

```
Alice, that was a great trick!
David, that was a great trick!
Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.
```

Это пример **логической ошибки**. Код имеет действительный синтаксис, но он не приводит к желаемому результату, потому что проблема кроется в его логике. Если некоторое действие должно повторяться для каждого элемента в списке, но выполняется только один раз, проверьте, не нужно ли добавить отступы в строке или нескольких строках кода.

ЛИШНИЕ ОТСТУПЫ

Если вы случайно поставите отступ в строке, в которой он не нужен, Python сообщит об этом:

```
message = "Hello Python world!»
❶  print(message)
```

Отступ команды print в точке ❶ не нужен, потому что эта строка не подчинена предшествующей; Python сообщает об ошибке:

```
File "hello_world.py", line 2
print(message)
^
IndentationError: unexpected indent
```

ЛИШНИЕ ОТСТУПЫ ПОСЛЕ ЦИКЛА

Если вы случайно снабдите отступом код, который должен выполняться *после* завершения цикла, то этот код будет выполнен для каждого элемента. Иногда Python выводит сообщение об ошибке, но часто дело ограничивается простой логической ошибкой. Например, что произойдет, если случайно снабдить отступом строку с выводом завершающего приветствия для группы фокусников?

```
magicians = ['alice', 'david', 'carolina’]
for magician in magicians:
    print(magician.title() + ", that was a great trick!")
    print("I can't wait to see your next trick, " + magician.title() + ".\n")
❶ print("Thank you everyone, that was a great magic show!")
```

Так как строка ❶ имеет отступ, сообщение будет продублировано для каждого фокусника в списке ❷:

```
Alice, that was a great trick!
I can't wait to see your next trick, Alice.

❷ ▪Thank you everyone, that was a great magic show!
David, that was a great trick!
I can't wait to see your next trick, David.

❷ ▪Thank you everyone, that was a great magic show!
Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.

❷ ▪Thank you everyone, that was a great magic show!
```

ПРОПУЩЕННОЕ ДВОЕТОЧИЕ

Двоеточие в конце команды for сообщает Python, что следующая строка является началом цикла.

```
magicians = ['alice', 'david', 'carolina’]
❶ for magician in magicians
    print(magician)
```

Если вы случайно забудете поставить двоеточие, как в примере ❶, произойдет синтаксическая ошибка, так как полученная команда нарушает правила языка. И хотя такие ошибки легко исправляются, найти их бывает достаточно трудно.

ЗАДАНИЕ 6

1. **Пицца:** вспомните по крайней мере три ваши любимые разновидности пиццы. Сохраните их в списке и используйте цикл for для вывода всех названий.
   - Измените цикл for так, чтобы вместо простого названия пиццы выводилось сообщение, включающее это название. Таким образом, для каждого элемента должна выводиться строка с простым текстом вида «I like pepperoni pizza» .
   - Добавьте в конец программы (после цикла for) строку с завершающим сообщением. Таким образом, вывод должен состоять из трех (и более) строк с названиями пиццы и дополнительного сообщения, скажем, «I really love pizza!»
   - Добавьте в список еще два вида пиццы и повторите цикл
2. **Животные**: создайте список из трех (и более) животных, обладающих общей характеристикой. Используйте цикл for для вывода названий всех животных.
   - Измените программу так, чтобы вместо простого названия выводилось сообщение, включающее это название, например «A dog would make a great pet» .
   - Добавьте в конец программы строку с описанием общего свойства. Например, можно вывести сообщение «Any of these animals would make a great pet!» .
   - Создайте второй список с именами владельцев этих домашних животных и при помощи использования цикла в цикле выведите поочерёдно информацию о том что у каждого из владельцев есть домашнее животное

СОЗДАНИЕ ЧИСЛОВЫХ СПИСКОВ

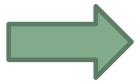
Необходимость хранения наборов чисел возникает в программах по многим причинам. Например, в компьютерной игре могут храниться координаты каждого персонажа на экране, таблицы рекордов и т. д. В программах визуализации данных пользователь почти всегда работает с наборами чисел: температурой, расстоянием, численностью населения, широтой/долготой и другими числовыми данными.

Списки идеально подходят для хранения наборов чисел, а Python предоставляет специальные средства для эффективной работы с числовыми списками. Достаточно один раз понять, как эффективно пользоваться этими средствами, и ваш код будет хорошо работать даже в том случае, если список содержит миллионы элементов.

ФУНКЦИЯ RANGE()

Функция range() упрощает построение числовых последовательностей. Например, с ее помощью можно легко вывести серию чисел:

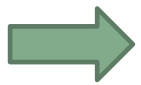
```
for value in range(1,5):
    print(value)
```



```
1
2
3
4
```

И хотя на первый взгляд может показаться, что он должен вывести числа от 1 до 5, на самом деле число 5 не выводится. В этом примере range() выводит только числа от 1 до 4. Перед вами еще одно проявление «смещения на 1», часто встречающегося в языках программирования. При выполнении функции range() Python начинает отсчет от первого переданного значения и прекращает его при достижении второго. Так как на втором значении происходит остановка, конец интервала (5 в данном случае) не встречается в выводе. Чтобы вывести числа от 1 до 5, используйте вызов range(1,6):

```
for value in range(1,6):
    print(value)
```

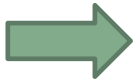


```
1
2
3
4
5
```

ИСПОЛЬЗОВАНИЕ RANGE() ДЛЯ СОЗДАНИЯ ЧИСЛОВОГО СПИСКА

Если вы хотите создать числовой список, преобразуйте результаты range() в список при помощи функции list(). Если заключить вызов range() в list(), то результат будет представлять собой список с числовыми элементами. В примере из предыдущего раздела числовая последовательность просто выводилась на экран. Тот же набор чисел можно преобразовать в список вызовом list():

```
numbers = list(range(1,6))
print(numbers)
```

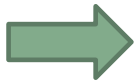


```
[1, 2, 3, 4, 5]
```

ИСПОЛЬЗОВАНИЕ RANGE() ДЛЯ СОЗДАНИЯ ЧИСЛОВОГО СПИСКА

Функция range() также может генерировать числовые последовательности, пропуская числа в заданном диапазоне. Например, построение списка четных чисел от 1 до 10 происходит так:

```
even_numbers = list(range(2,11,2))
print(even_numbers)
```



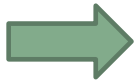
```
[2, 4, 6, 8, 10]
```

В этом примере функция range() начинает со значения 2, а затем увеличивает его на 2. Приращение 2 последовательно применяется до тех пор, пока не будет достигнуто или пройдено конечное значение 11, после чего выводится результат.

ИСПОЛЬЗОВАНИЕ RANGE() ДЛЯ СОЗДАНИЯ ЧИСЛОВОГО СПИСКА

С помощью функции range() можно создать практически любой диапазон чисел. Например, как бы вы создали список квадратов всех целых чисел от 1 до 10? В языке Python операция возведения в степень обозначается двумя звездочками (**). Один из возможных вариантов выглядит так:

```
❶ squares = []
❷ for value in range(1,11):
❸     square = value**2
❹     squares.append(square)
❺ print(squares)
```



```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Сначала в точке 1 создается пустой список с именем squares. В точке 2 вы приказываете Python перебрать все значения от 1 до 10 при помощи функции range(). В цикле текущее значение возводится во вторую степень, а результат сохраняется в переменной square в точке 3. В точке 4 каждое новое значение square присоединяется к списку squares. Наконец, после завершения цикла список квадратов выводится в точке 5.

ПРОСТАЯ СТАТИСТИКА С ЧИСЛОВЫМИ СПИСКАМИ

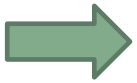
Некоторые функции Python предназначены для работы с числовыми списками. Например, вы можете легко узнать минимум, максимум и сумму числового списка:

```
>>> digits = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
>>> min(digits)
0
>>> max(digits)
9
>>> sum(digits)
45
```

ГЕНЕРАТОРЫ СПИСКОВ

Описанный выше способ генерирования списка squares состоял из четырех строк кода. *Генератор списка* (list comprehension) позволяет сгенерировать тот же список всего в одной строке. Генератор списка объединяет цикл for и создание новых элементов в одну строку и автоматически присоединяет к списку все новые элементы. В следующем примере список квадратов, знакомый вам по предыдущим примерам, строится с использованием генератора списка:

```
squares = [value**2 for value in range(1,11)]
print(squares)
```



```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Чтобы использовать этот синтаксис, начните с содержательного имени списка, например squares. Затем откройте квадратные скобки и определите выражение для значений, которые должны быть сохранены в новом списке. В данном примере это выражение value**2, которое возводит значение во вторую степень. Затем напишите цикл for для генерирования чисел, которые должны передаваться выражению, и закройте квадратные скобки. Цикл for в данном примере — for value in range(1,11) — передает значения с 1 до 10 выражению value**2. Обратите внимание на отсутствие двоеточия в конце команды for. Результатом будет уже знакомый вам список квадратов.

ЗАДАНИЕ 7

1. **Считаем до 20**: используйте цикл for для вывода чисел от 1 до 20 включительно.
2. **Тысячи:** создайте список чисел от 1 до 10 000, затем воспользуйтесь циклом for для вывода чисел. (Если вывод занимает слишком много времени, остановите его нажатием Ctrl+C или закройте окно вывода).
3. **Суммирование чисел**: создайте список чисел от 1 до 10 000, затем воспользуйтесь функциями min() и max() и убедитесь в том, что список действительно начинается с 1 и заканчивается 1 000 000. Вызовите функцию sum() и посмотрите, насколько быстро Python сможет просуммировать миллион чисел.
4. **Нечетные числа**: воспользуйтесь третьим аргументом функции range() для создания списка нечетных чисел от 1 до 20. Выведите все числа в цикле for.
5. **Тройки**: создайте список чисел, кратных 3, в диапазоне от 3 до 30. Выведите все числа своего списка в цикле for.
6. **Кубы**: результат возведения числа в третью степень называется кубом. Например, куб 2 записывается в языке Python в виде 2**3. Создайте список первых 10 кубов (то есть кубов всех целых чисел от 1 до 10) и выведите значения всех кубов в цикле for.

РАБОТА С ЧАСТЬЮ СПИСКА

В главе 3 вы узнали, как обращаться к отдельным элементам списка, а в этой главе мы занимались перебором всех элементов списка. Также можно работать с конкретным подмножеством элементов списка; в Python такие подмножества называются *срезами* (slices).

СОЗДАНИЕ СРЕЗА

Чтобы создать срез списка, следует задать индексы первого и последнего элементов, с которыми вы намереваетесь работать. Как и в случае с функцией range(), Python останавливается на элементе, предшествующем второму индексу. Скажем, чтобы вывести первые три элемента списка, запросите индексы с 0 по 3, и вы получите элементы 0, 1 и 2. В следующем примере используется список игроков команды:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli’]
❶ print(players[0:3])
```

В точке ❶ выводится срез списка, включающий только первых трех игроков. Вывод сохраняет структуру списка, но включает только первых трех игроков:

```
['charles', 'martina', 'michael']
```

СОЗДАНИЕ СРЕЗА

Подмножество может включать любую часть списка. Например, чтобы ограничиться вторым, третьим и четвертым элементами списка, срез начинается с индекса 1 и заканчивается на индексе 4:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[1:4])
```

На этот раз срез начинается с элемента 'martina' и заканчивается элементом 'florence':

```
['martina', 'michael', 'florence']
```

Если первый индекс среза не указан, то Python автоматически начинает срез от начала списка:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[:4])
```

```
['charles', 'martina', 'michael', 'florence']
```

СОЗДАНИЕ СРЕЗА

Аналогичный синтаксис работает и для срезов, включающих конец списка. Например, если вам нужны все элементы с третьего до последнего, начните с индекса 2 и не указывайте второй индекс:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[2:])
```

```
['michael', 'florence', 'eli']
```

Этот синтаксис позволяет вывести все элементы от любой позиции до конца списка независимо от его длины. Вспомните, что отрицательный индекс возвращает элемент, находящийся на заданном расстоянии от конца списка; следовательно, вы можете получить любой срез от конца списка. Например, чтобы отобрать последних трех игроков, используйте срез **players[-3:]**:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[-3:])
```

КОПИРОВАНИЕ СПИСКА

Чтобы скопировать список, создайте срез, включающий весь исходный список без указания первого и второго индекса ([:]). Эта конструкция создает срез, который начинается с первого элемента и завершается последним; в результате создается копия всего списка.

Представьте, что вы создали список своих любимых блюд и теперь хотите создать отдельный список блюд, который нравится Вашему другу. Пока вашему другу нравятся все блюда из нашего списка, поэтому вы можете создать другой список простым копированием нашего:

```
my_foods = [‘pizza', ‘cake', ‘borsch', ‘salad']
friends_foods = my_foods[:]
print (my_foods)
print (friends_foods)

[‘pizza', ‘cake', ‘borsch', ‘salad']
[‘pizza', ‘cake', ‘borsch', ‘salad']
```

ПЕРЕБОР СОДЕРЖИМОГО СРЕЗА

Если вы хотите перебрать элементы, входящие в подмножество элементов, используйте срез в цикле for. В следующем примере программа перебирает первых трех игроков и выводит их имена:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli’]
print("Here are the first three players on my team:")
❶ for player in players[:3]:
    print(player.title())
```

Вместо того чтобы перебирать весь список игроков ❶, Python ограничивается первыми тремя именами:

```
Here are the first three players on my team:
Charles
Martina
Michael
```

ЗАДАНИЕ 8

1. **Срезы:** добавьте в конец одной из программ, написанных ранее, фрагмент, который делает следующее:
   - Выводит сообщение «The first three items in the list are:», а затем использует срез для вывода первых трех элементов из списка.
   - Выводит сообщение «Three items from the middle of the list are:», а затем использует срез для вывода первых трех элементов из середины списка.
   - Выводит сообщение «The last three items in the list are:», а затем использует срез для вывода последних трех элементов из списка.
2. **Моя пицца, твоя пицца**: начните с программы из упражнения 6. Создайте копию списка с видами пиццы, присвойте ему имя friend_pizzas. Затем сделайте следующее:
   - Добавьте новую пиццу в исходный список.
   - Добавьте другую пиццу в список friend_pizzas.
   - Докажите, что в программе существуют два разных списка. Выведите сообщение «My favorite pizzas are:», а затем первый список в цикле for. Выведите сообщение «My friend's favorite pizzas are:», а затем второй список в цикле for. Убедитесь в том, что каждая новая пицца находится в соответствующем списке.

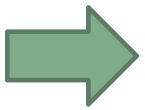
ЧАСТЬ IV

КОМАНДЫ IF

ПРОСТОЙ ПРИМЕР

Допустим, у вас имеется список машин, и вы хотите вывести название каждой машины. Названия большинства машин должны записываться с капитализацией (первая буква в верхнем регистре, остальные в нижнем). С другой стороны, значение 'bmw' должно записываться в верхнем регистре. Следующий код перебирает список названий машин и ищет в нем значение 'bmw'. Для всех элементов, содержащих значение 'bmw', значение выводится в верхнем регистре:

```
cars = ['audi', 'bmw', 'subaru', 'toyota’]
for car in cars:
    if car == 'bmw’:
        print(car.upper())
    else:
        print(car.title())
```



```
Audi
BMW
Subaru
Toyota
```

Цикл в этом примере I сначала проверяет, содержит ли car значение 'bmw'. Если проверка дает положительный результат, то значение выводится в верхнем регистре. Если car содержит все что угодно, кроме 'bmw', то при выводе значения применяется капитализация.

ПРОВЕРКА УСЛОВИЙ

В каждой команде if центральное место занимает выражение, результатом которого является логическая истина (True) или логическая ложь (False); это выражение называется условием. В зависимости от результата проверки Python решает, должен ли выполняться код в команде if. Если результат условия равен True, то Python выполняет код, следующий за командой if.

ПРОВЕРКА РАВЕНСТВА

Во многих условиях текущее значение переменной сравнивается с конкретным значением, интересующим вас. Простейшее условие проверяет, равно ли значение переменной конкретной величине:

```
❶ >>> car = 'bmw'
❷ >>> car == 'bmw'
True
```

В строке 1 переменной car присваивается значение 'bmw'; операция выполняется одним знаком =, как вы уже неоднократно видели. Строка 2 проверяет, равно ли значение car строке 'bmw'; для проверки используется двойной знак равенства (==). Этот оператор возвращает True, если значения слева и справа от оператора равны; если же значения не совпадают, оператор возвращает False. В нашем примере значения совпадают, поэтому Python возвращает True.

ПРОВЕРКА РАВЕНСТВА БЕЗ УЧЕТА РЕГИСТРА

В языке Python проверка равенства выполняется с учетом регистра. Например, два значения с разным регистром символов равными не считаются:

```
>>> car = 'Audi’
>>> car == 'audi’
False
```

Если регистр символов важен, такое поведение приносит пользу. Но если проверка должна выполняться на уровне символов без учета регистра, преобразуйте значение переменной к нижнему регистру перед выполнением сравнения:

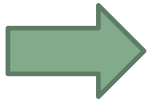
```
>>> car = 'Audi'
>>> car.lower() == 'audi’
True
```

ПРОВЕРКА НЕРАВЕНСТВА

Если вы хотите проверить, что два значения *различны*, используйте комбинацию из восклицательного знака и знака равенства (!=). Восклицательный знак представляет отрицание, как и во многих языках программирования.

Для знакомства с оператором неравенства мы воспользуемся другой командой if. В переменной хранится заказанное дополнение к пицце; если клиент не заказал анчоусы (anchovies), программа выводит сообщение:

```
requested_topping = 'mushrooms'
❶ if requested_topping != 'anchovies':
    print("Hold the anchovies!")
```



```
Hold the anchovies!
```

Строка 1 сравнивает значение requested_topping со значением 'anchovies'. Если эти два значения не равны, Python возвращает True и выполняет код после команды if. Если два значения равны, Python возвращает False и не выполняет код после команды if. Так как значение requested_topping отлично от 'anchovies', команда print не будет выполнена.

СРАВНЕНИЯ ЧИСЕЛ

Проверка числовых значений достаточно прямолинейна. Например, следующий код проверяет, что переменная age равна 18:

```
>>> age = 18
>>> age == 18
True
```

Также можно проверить условие неравенства двух чисел. Например, следующий код выводит сообщение, если значение переменной answer отлично от ожидаемого:

```
answer = 17
❶ if answer != 42:
    print("That is not the correct answer. Please try again!")
```

Условие 1 выполняется, потому что значение answer (17) не равно 42. Так как условие истинно, блок с отступом выполняется:

```
That is not the correct answer. Please try again!
```

СРАВНЕНИЯ ЧИСЕЛ

В условные команды также можно включать всевозможные математические сравнения: меньше, меньше или равно, больше, больше или равно:

```
>>> age = 19
>>> age < 21
True
>>> age <= 21
True
>>> age > 21
False
>>> age >= 21
False
```

ПРОВЕРКА НЕСКОЛЬКИХ УСЛОВИЙ

Иногда требуется проверить несколько условий одновременно. Например, в некоторых случаях для выполнения действия бывает нужно, чтобы истинными были сразу два условия; в других случаях достаточно, чтобы истинным было хотя бы одно из двух условий. Ключевые слова and и or помогут вам в подобных ситуациях.

ИСПОЛЬЗОВАНИЕ AND ДЛЯ ПРОВЕРКИ НЕСКОЛЬКИХ УСЛОВИЙ

Чтобы проверить, что два условия истинны одновременно, объедините их ключевым словом and; если оба условия истинны, то и все выражение тоже истинно. Если хотя бы одно (или оба) условия ложны, то и результат всего выражения равен False. Например, чтобы убедиться в том, что каждому из двух людей больше 21 года, используйте следующую проверку:

```
❶ >>> age_0 = 22
  >>> age_1 = 18
❷ >>> age_0 >= 21 and age_1 >= 21
  False
❸ >>> age_1 = 22
  >>> age_0 >= 21 and age_1 >= 21
  True
```

В точке 1 определяются две переменные, age_0 и age_1. В точке 2 программа проверяет, что оба значения равны 21 и более. Левое условие выполняется, а правое нет, поэтому все условное выражение дает результат False. В точке 3 переменной age_1 присваивается значение 22. Теперь значение age_1 больше 21; обе проверки проходят, а все условное выражение дает истинный результат.

Чтобы код лучше читался, отдельные условия можно заключить в круглые скобки, но это не обязательно. С круглыми скобками проверка может выглядеть так:

```
(age_0 >= 21) and (age_1 >= 21)
```

ИСПОЛЬЗОВАНИЕ OR ДЛЯ ПРОВЕРКИ НЕСКОЛЬКИХ УСЛОВИЙ

Ключевое слово or тоже позволяет проверить несколько условий, но результат общей проверки является истинным в том случае, когда истинно хотя бы одно или оба условия. Ложный результат достигается только в том случае, если оба отдельных условия ложны. Вернемся к примеру с возрастом, но на этот раз проверим, что хотя бы одна из двух переменных больше 21:

```
❶ >>> age_0 = 22
  >>> age_1 = 18
❷ >>> age_0 >= 21 or age_1 >= 21
  True
❸ >>> age_0 = 18
  >>> age_0 >= 21 or age_1 >= 21
  False
```

Как и в предыдущем случае, в точке 1 определяются две переменные. Так как условие для age_0 в точке 2 истинно, все выражение также дает истинный результат. Затем значение age_0 уменьшается до 18. При проверке 3 оба условия оказываются ложными, и общий результат всего выражения тоже ложен.

ПРОВЕРКА ВХОЖДЕНИЯ ЗНАЧЕНИЙ В СПИСОК

Иногда бывает важно проверить, содержит ли список некоторое значение, прежде чем выполнять действие. Например, перед завершением регистрации нового пользователя на сайте можно проверить, существует ли его имя в списке имен действующих пользователей, или в картографическом проекте определить, входит ли передаваемое место в список известных мест на карте. Чтобы узнать, присутствует ли заданное значение в списке, воспользуйтесь ключевым словом in. Допустим, вы пишете программу для пиццерии. Вы создали список дополнений к пицце, заказанных клиентом, и хотите проверить, входят ли некоторые дополнения в этот список.

```
>>> requested_toppings = ['mushrooms', 'onions', 'pineapple’]
❶ >>> 'mushrooms' in requested_toppings
True
❷ >>> 'pepperoni' in requested_toppings
False
```

В точках 1 и 2 ключевое слово in приказывает Python проверить, входят ли значения 'mushrooms' и 'pepperoni' в список requested_toppings. Это весьма полезно, потому что вы можете создать список значений, критичных для вашей программы, а затем легко проверить, присутствует ли проверяемое значение в списке.

ПРОВЕРКА ОТСУТСТВИЯ ЗНАЧЕНИЯ В СПИСКЕ

В других случаях программа должна убедиться в том, что значение *не входит* в список. Для этого используется ключевое слово not. Для примера рассмотрим список пользователей, которым запрещено писать комментарии на форуме. Прежде чем разрешить пользователю отправку комментария, можно проверить, не был ли пользователь включен в «черный список»:

```
banned_users = ['andrew', 'carolina', 'david']
user = 'marie'
❶ if user not in banned_users:
    print(user.title() + ", you can post a response if you wish.")
```

Строка 1 достаточно четко читается: если пользователь не входит в «черный список» banned_users, то Python возвращает True и выполняет строку с отступом. Пользователь 'marie' в этот список не входит, поэтому программа выводит соответствующее сообщение:

```
Marie, you can post a response if you wish.
```

ЛОГИЧЕСКИЕ ВЫРАЖЕНИЯ

В процессе изучения программирования вы рано или поздно услышите термин «логическое выражение». По сути это всего лишь другое название для проверки условия. Результат логического выражения равен True или False, как и результат условного выражения после его вычисления. Логические выражения часто используются для проверки некоторых условий — например, запущена ли компьютерная игра или разрешено ли пользователю редактирование некоторой информации на сайте:

```
game_active = True
can_edit = False
```

Логические выражения предоставляют эффективные средства для контроля состояния программы или определенного условия, играющего важную роль в вашей программе.

ЗАДАНИЕ 9

1. **Проверка условий:** напишите последовательность условий. Выведите описание каждой проверки и ваш прогноз относительно ее результата. Код должен выглядеть примерно так:

```
car = 'subaru'

print("Is car == 'subaru'? I predict True.")

print(car == 'subaru')

print("\nIs car == 'audi'? I predict False.")

print(car == 'audi')
```

   - Внимательно просмотрите результаты. Убедитесь в том, что вы понимаете, почему результат каждой строки равен True или False.
   - Создайте как минимум 10 условий. Не менее 5 должны давать результат True, а не менее 5 других — результат False.

ЗАДАНИЕ 9
(ПРОДОЛЖЕНИЕ)

2. **Больше условий:** количество условий не ограничивается 10. Попробуйте написать другие условия и включить их в предыдущий тест. Программа должна выдавать по крайней мере один истинный и один ложный результат для следующих видов проверок:
   - Проверка равенства и неравенства строк.
   - Проверки с использованием функции lower().
   - Числовые проверки равенства и неравенства, условий «больше», «меньше», «больше или равно», «меньше или равно».
   - Проверки с ключевым словом and и or.
   - Проверка вхождения элемента в список.
   - Проверка отсутствия элемента в списке.

ПРОСТЫЕ КОМАНДЫ IF

Простейшая форма команды if состоит из одного условия и одного действия:

```
if условие:
    действие
```

В первой строке размещается условие, а в блоке с отступом — практически любое действие. Если условие истинно, то Python выполняет код в блоке после команды if, а если ложно, этот код игнорируется. Допустим, имеется переменная, представляющая возраст человека. Следующий код проверяет, что этот возраст достаточен для голосования:

```
age = 19
❶ if age >= 18:
❷   print("You are old enough to vote!")
```

В точке 1 Python проверяет, что значение переменной age больше или равно 18. В таком случае выполняется команда print 2 в строке с отступом:

```
You are old enough to vote!
```

ПРОСТЫЕ КОМАНДЫ IF

Отступы в командах if играют ту же роль, что и в циклах for. Если условие истинно, то все строки с отступом после команды if выполняются, а если ложно — весь блок с отступом игнорируется. Блок команды if может содержать сколько угодно строк. Добавим еще одну строку для вывода дополнительного сообщения в том случае, если возраст достаточен для голосования:

```
age = 19
if age >= 18:
    print("You are old enough to vote!")
    print("Have you registered to vote yet?")
```

Условие выполняется, а обе команды print снабжены отступом, поэтому выводятся оба сообщения:

```
You are old enough to vote!
Have you registered to vote yet?
```

Если значение age меньше 18, программа ничего не выводит.

КОМАНДЫ IF-ELSE

Часто в программе необходимо выполнить одно действие в том случае, если условие истинно, и другое действие, если оно ложно. С синтаксисом if-else это возможно. Блок if-else в целом похож на команду if, но секция else определяет действие или набор действий, выполняемых при неудачной проверке. В следующем примере выводится то же сообщение, которое выводилось ранее, если возраст достаточен для голосования, но на этот раз при любом другом возрасте выводится другое сообщение:

```
age = 17
❶ if age >= 18:
    print("You are old enough to vote!") print("Have you registered to vote yet?")
❷ else:
    print("Sorry, you are too young to vote.")
    print("Please register to vote as soon as you turn 18!")
```

Если условие 1 истинно, то выполняется первый блок с командами print. Если же условие ложно, выполняется блок else в точке 2. Так как значение age на этот раз меньше 18, условие оказывается ложным, и выполняется код в блоке else:

```
Sorry, you are too young to vote. Please
register to vote as soon as you turn 18!
```

ЦЕПОЧКИ IF-ELIF-ELSE

Нередко в программе требуется проверять более двух возможных ситуаций; для таких ситуаций в Python предусмотрен синтаксис if-elif-else. Python выполняет только один блок в цепочке if-elif-else. Все условия проверяются по порядку до тех пор, пока одно из них не даст истинный результат. Далее выполняется код, следующий за этим условием, а все остальные проверки Python пропускает.

Во многих реальных ситуациях существуют более двух возможных результатов. Представьте себе парк аттракционов, который взимает разную плату за вход для разных возрастных групп:

- Для посетителей младше 4 лет вход бесплатный.
- Для посетителей от 4 до 18 лет билет стоит $5.
- Для посетителей от 18 лет и старше билет стоит $10.

Как использовать команду if для определения платы за вход? Следующий код определяет, к какой возрастной категории относится посетитель, и выводит сообщение со стоимостью билета:

ЦЕПОЧКИ IF-ELIF-ELSE

```
age = 12
❶ if age < 4:
    print("Your admission cost is $0.")
❷ elif age < 18:
    print("Your admission cost is $5.")
❸ else:
    print("Your admission cost is $10.")
```

Условие if в точке 1 проверяет, что возраст посетителя меньше 4 лет. Если условие истинно, то программа выводит соответствующее сообщение, и Python пропускает остальные проверки. Строка elif в точке 2 в действительности является еще одной проверкой if, которая выполняется только в том случае, если предыдущая проверка завершилась неудачей. В этом месте цепочки известно, что возраст посетителя не меньше 4 лет, потому что первое условие было ложным. Если посетителю меньше 18 лет, программа выводит соответствующее сообщение, и Python пропускает блок else. Если ложны оба условия — if и elif, то Python выполняет код в блоке else в точке 3.В данном примере условие 1 дает ложный результат, поэтому его блок не выполняется. Однако второе условие оказывается истинным (12 меньше 18), поэтому код будет выполнен. Вывод состоит из одного сообщения с ценой билета:

```
Your admission cost is $5.
```

ЦЕПОЧКИ IF-ELIF-ELSE

Вместо того чтобы выводить сообщение с ценой билета в блоках if-elif-else, лучше использовать другое, более компактное решение: присвоить цену в цепочке if-elif-else, а затем добавить одну команду print после выполнения цепочки:

```
age = 12

if age < 4:
❶     price = 0
elif age < 18:
❷     price = 5
else:
❸     price = 10

❹ print("Your admission cost is $" + str(price) + ".")
```

Строки 1, 2 и 3 присваивают значение price в зависимости от значения age, как и в предыдущем примере. После присваивания цены в цепочке if-elif-else отдельная команда print без отступа 4 использует это значение для вывода сообщения с ценой билета.

СЕРИИ БЛОКОВ ELIF

Код может содержать сколько угодно блоков elif. Например, если парк аттракционов введет особую скидку для пожилых посетителей, вы можете добавить в свой код еще одну проверку для определения того, распространяется ли скидка на текущего посетителя. Допустим, посетители с возрастом 65 и выше платят половину от обычной цены билета, или $5:

```
age = 12
if age < 4:
    price = 0
elif age < 18:
    price = 5
❶ elif age < 65:
    price = 10
❷ else:
    price = 5
print("Your admission cost is $" + str(price) + ".")
```

Большая часть кода осталась неизменной. Второй блок elif в точке ❶ теперь убеждается в том, что посетителю меньше 65 лет, прежде чем назначить ему полную цену билета $10. Обратите внимание: значение, присвоенное в блоке else ❷, должно быть заменено на $5, потому что до этого блока доходят только посетители с возрастом 65 и выше.

ОТСУТСТВИЕ БЛОКА ELSE

Python не требует, чтобы цепочка if-elif непременно завершалась блоком else. Иногда блок else удобен; в других случаях бывает нагляднее использовать дополнительную секцию elif для обработки конкретного условия:

```
age = 12
if age < 4:
price = 0
elif age < 18:
price = 5
elif age < 65:
price = 10
❶ elif age >= 65:
price = 5
print("Your admission cost is $" + str(price) + ".")
```

Блок elif в точке 1 назначает цену $5, если возраст посетителя равен 65 и выше; смысл такого кода более понятен, чем у обобщенного блока else. С таким изменением выполнение каждого блока возможно только при истинности конкретного условия.

ПРОВЕРКА НЕСКОЛЬКИХ УСЛОВИЙ

Цепочки if-elif-else эффективны, но они подходят только в том случае, если истинным должно быть только одно условие. Как только Python находит выполняющееся условие, все остальные проверки пропускаются. Такое поведение достаточно эффективно, потому что оно позволяет проверить одно конкретное условие. Однако иногда бывает важно проверить все условия, представляющие интерес. В таких случаях следует применять серии простых команд if без блоков elif или else. Такое решение уместно, когда истинными могут быть сразу несколько условий и вы хотите отреагировать на все истинные условия. Вернемся к примеру с пиццей. Если кто-то закажет пиццу с двумя дополнениями, программа должна обработать оба дополнения:

```
❶ requested_toppings = ['mushrooms', 'extra cheese']
❷ if 'mushrooms' in requested_toppings:
    print("Adding mushrooms.")
❸ if 'pepperoni' in requested_toppings:
    print("Adding pepperoni.")
❹ if 'extra cheese' in requested_toppings:
      print("Adding extra cheese.")
print("\nFinished making your pizza!")
```

Обработка начинается в точке 1 со списка, содержащего заказанные дополнения. Команды if в точке 2 и 3 проверяют, включает ли заказ конкретные дополнения — грибы и пепперони, и если включает — выводят подтверждающее сообщение. Проверка в точке 3 реализована простой командой if, а не elif или else, поэтому условие будет проверяться независимо от того, было ли предыдущее условие ис-инным или ложным. Код в точке 4 проверяет, была ли заказана дополнительная порция сыра, независимо от результата первых двух проверок. Эти три независимых условия проверяются при каждом выполнении программы.

Так как в этом коде проверяются все возможные варианты дополнений, в заказ будут включены два дополнения из трех:

```
Adding mushrooms.
Adding extra cheese.
Finished making your pizza!
```

ЗАДАНИЕ 10

1. **Цвета 1:** представьте, что в вашей компьютерной игре только что был подбит корабль пришельцев. Создайте переменную с именем alien_color и присвойте ей значение ‘green’, ‘yellow’ или ‘red’:
   - Напишите команду if для проверки того, что переменная содержит значение ‘green’. Если условие истинно, выведите сообщение о том, что игрок только что заработал 5 очков.
   - Напишите одну версию программы, в которой условие if выполняется, и другую версию, в которой оно не выполняется. (Во второй версии никакое сообщение выводиться не должно).
2. **Цвета 2**: выберите цвет, как это было сделано в предыдущем упражнении, и напишите цепочку if-else:
   - Напишите команду if для проверки того, что переменная содержит значение ‘green’. Если условие истинно, выведите сообщение о том, что игрок только что заработал 5 очков.
   - Если переменная содержит любое другое значение, выведите сообщение о том, что игрок только что заработал 10 очков.
   - Напишите одну версию программы, в которой выполняется блок if, и другую версию, в которой выполняется блок else.

ЗАДАНИЕ 10
(ПРОДОЛЖЕНИЕ)

1. **Цвета 3:** преобразуйте цепочку if-else из предыдущего упражнения в цепочку if-elif-else:
   - Если переменная содержит значение 'green', выведите сообщение о том, что игрок только что заработал 5 очков.
   - Если переменная содержит значение 'yellow', выведите сообщение о том, что игрок только что заработал 10 очков.
   - Если переменная содержит значение 'red', выведите сообщение о том, что игрок только что заработал 15 очков.
   - Напишите три версии программы и проследите за тем, чтобы для каждого цвета пришельца выводилось соответствующее сообщение.
2. **Периоды жизни:** напишите цепочку if-elif-else для определения периода жизни человека. Присвойте значение переменной age, а затем выведите сообщение:
   - Если значение меньше 2 — младенец.
   - Если значение больше или равно 2, но меньше 4 — малыш.
   - Если значение больше или равно 4, но меньше 13 — ребенок.
   - Если значение больше или равно 13, но меньше 20 — подросток.
   - Если значение больше или равно 20, но меньше 40 — молодежь.
   - Если значение больше или равно 40, но меньше 65 — взрослый.
   - Если значение больше или равно 65 — пожилой человек.

ЗАДАНИЕ 10
(ПРОДОЛЖЕНИЕ)

1. **Любимый фрукт:** составьте список своих любимых фруктов. Напишите серию независимых команд if для проверки того, присутствуют ли некоторые фрукты в списке:
   - Создайте список трех своих любимых фруктов и назовите его favorite_fruits.
   - Напишите пять команд if. Каждая команда должна проверять, входит ли определенный тип фрукта в список. Если фрукт входит в список, блок if должен выводить сообщение вида «You really like bananas!».

ИСПОЛЬЗОВАНИЕ КОМАНД IF СО СПИСКАМИ

Объединение команд if со списками открывает ряд интересных возможностей. Например, вы можете отслеживать специальные значения, для которых необходима особая обработка по сравнению с другими значениями в списке, или эффективно управлять изменяющимися условиями — например, наличием некоторых блюд в ресторане. Также объединение команд if со списками помогает продемонстрировать, что ваш код корректно работает во всех возможных ситуациях.

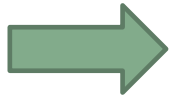
ПРОВЕРКА СПЕЦИАЛЬНЫХ ЗНАЧЕНИЙ

Вернемся к примеру с пиццерией. Программа выводит сообщение каждый раз, когда пицца снабжается дополнением в процессе приготовления. Код этого действия можно записать чрезвычайно эффективно: нужно создать список дополнений, заказанных клиентом, и использовать цикл для перебора всех заказанных дополнений:

```
requested_toppings = ['mushrooms', 'green peppers', 'extra cheese']

for requested_topping in requested_toppings:
    print("Adding " + requested_topping + ".")

print("\nFinished making your pizza!")
```



```
Adding mushrooms.
Adding green peppers.
Adding extra cheese.

Finished making your pizza!
```

Вывод достаточно тривиален, поэтому код сводится к простому циклу for.

ПРОВЕРКА СПЕЦИАЛЬНЫХ ЗНАЧЕНИЙ

А если в пиццерии вдруг кончится зеленый перец? Команда if в цикле for может правильно обработать эту ситуацию:

```
requested_toppings = ['mushrooms', 'green peppers', 'extra cheese']

for requested_topping in requested_toppings:
❶    if requested_topping == 'green peppers':
         print("Sorry, we are out of green peppers right now.")
❷    else:
print("Adding " + requested_topping + ".")

print("\nFinished making your pizza!")
```

```
Adding mushrooms.
Sorry, we are out of green peppers right now.
Adding extra cheese.

Finished making your pizza!
```

На этот раз программа проверяет каждый заказанный элемент перед добавлением его к пицце. В точке 1 программа проверяет, заказал ли клиент зеленый перец, и если заказал — выводит сообщение о том, что этого дополнения нет. Блок **else** в точке 2 гарантирует, что все другие дополнения будут включены в заказ.

ПРОВЕРКА НАЛИЧИЯ ЭЛЕМЕНТОВ В СПИСКЕ

Для всех списков, с которыми мы работали до сих пор, действовало одно простое предположение: мы считали, что в каждом списке есть хотя бы один элемент. Скоро мы предоставим пользователю возможность вводить информацию, хранящуюся в списке, поэтому мы уже не можем предполагать, что при каждом выполнении цикла в списке есть хотя бы один элемент. В такой ситуации перед выполнением цикла for будет полезно проверить, есть ли в списке хотя бы один элемент. Проверим, есть ли элементы в списке заказанных дополнений, перед изготовлением пиццы. Если список пуст, программа предлагает пользователю подтвердить, что он хочет базовую пиццу без дополнений. Если список не пуст, пицца готовится так же, как в предыдущих примерах:

```
❶ requested_toppings = []

❷ if requested_toppings:
    for requested_topping in requested_toppings:
    print("Adding " + requested_topping + ".")
    print("\nFinished making your pizza!")
❸ else:
        print("Are you sure you want a plain pizza?")
```

```
Are you sure you want a plain pizza?
```

На этот раз мы начинаем с пустого списка заказанных дополнений в точке 1. Вместо того чтобы сразу переходить к циклу for, программа выполняет проверку в точке 2. Когда имя списка используется в условии if, Python возвращает True, если список содержит хотя бы один элемент; если список пуст, возвращается значение False. Если requested_toppings проходит проверку условия, выполняется тот же цикл for, который мы использовали в предыдущем примере. Если же условие ложно, то программа выводит сообщение, которое предлагает клиенту подтвердить, действительно ли он хочет получить базовую пиццу без дополнений 3.

МНОЖЕСТВЕННЫЕ СПИСКИ

Посетители способны заказать что угодно, особенно когда речь заходит о дополнениях к пицце. Что если клиент захочет положить на пиццу картофель фри? Списки и команды if позволят вам убедиться в том, что входные данные имеют смысл, прежде чем обрабатывать их. Давайте проверим наличие нестандартных дополнений перед тем, как готовить пиццу. В следующем примере определяются два списка. Первый список содержит перечень доступных дополнений, а второй — список дополнений, заказанных клиентом. На этот раз каждый элемент из requested_toppings проверяется по списку доступных дополнений перед добавлением в пиццу:

```
❶ available_toppings = ['mushrooms', 'olives', 'green peppers',
                        'pepperoni', 'pineapple', 'extra cheese']

❷ requested_toppings = ['mushrooms', 'french fries', 'extra cheese']

❸ for requested_topping in requested_toppings:
❹     if requested_topping in available_toppings:
          print("Adding " + requested_topping + ".")
❺     else:
          print("Sorry, we don't have " + requested_topping + ".")

  print("\nFinished making your pizza!")
```

```
Adding mushrooms.
Sorry, we don't have french fries.
Adding extra cheese.

Finished making your pizza!
```

В точке 1 определяется список доступных дополнений к пицце. Стоит заметить, что если в пиццерии используется постоянный ассортимент дополнений, этот список можно реализовать в виде кортежа. В точке 2 создается список дополнений, заказанных клиентом. Обратите внимание на необычный заказ 'french fries'. В точке 3 программа перебирает список заказанных дополнений. Внутри цикла программа сначала проверяет, что каждое заказанное дополнение присутствует в списке доступных дополнений 4 . Если дополнение доступно, оно добавляется в пиццу. Если заказанное дополнение не входит в список, выполняется блок else 5. Блок else выводит сообщение о том, что дополнение недоступно.

ЗАДАНИЕ 11

1. **Hello Admin**: создайте список из пяти и более имен пользователей, включающий имя ‘admin’. Представьте, что вы пишете код, который выводит приветственное сообщение для каждого пользователя после его входа на сайт. Переберите элементы списка и выведите сообщение для каждого пользователя:
   - Для пользователя с именем 'admin’ выведите особое сообщение — например: «Hello admin, would you like to see a status report?»
   - В остальных случаях выводите универсальное приветствие — например: «Hello Eric, thank you for logging in again».
2. Проверьте наличие элементов в списке при помощи команды if:
   - Если список пуст, выведите сообщение: «We need to find some users!»
   - Удалите из списка все имена пользователей и убедитесь в том, что программа выводит правильное сообщение .

ЗАДАНИЕ 11
(ПРОДОЛЖЕНИЕ)

1. **Проверка имен пользователей**: выполните следующие действия для создания программы, моделирующей проверку уникальности имен пользователей:
   - Создайте список current_users, содержащий пять и более имен пользователей.
   - Создайте другой список new_users, содержащий пять и более имен пользователей. Убедитесь в том, что одно или два новых имени также присутствуют в списке current_ users.
   - Переберите список new_users и для каждого имени в этом списке проверьте, было ли оно использовано ранее. Если имя уже использовалось, выведите сообщение о том, что пользователь должен выбрать новое имя. Если имя не использовалось, выведите сообщение о его доступности.
   - Проследите за тем, чтобы сравнение выполнялось без учета регистра символов. Если имя 'John' уже используется, в регистрации имени 'JOHN' следует отказать.

- **Порядковые числительные**: порядковые числительные в английском языке заканчиваются суффиксом th (кроме 1st, 2nd и 3rd):
  - Сохраните числа от 1 до 9 в списке.
  - Переберите элементы списка.
  - Используйте цепочку if-elif-else в цикле для вывода правильного окончания числи- тельного для каждого числа. Программа должна выводить числительные «1st 2nd 3rd 4th 5th 6th 7th 8th 9th», причем каждый результат должен располагаться в отдельной строке.