Модуль 1

Начало работы

Установка

Последняя версия Python доступна по адресу https://www.python.org/downloads/.

Пользователям Linux и Mac

Скорее всего, Python в вашей системе уже установлен. Чтобы это проверить, откройте терминал и наберите:

```
$ python3 -V
```

Если вы видите информацию о версии, значит Python у вас уже установлен.

Пользователям Windows

Установка производится так же, как и для любых других программ для Windows.

Осторожно Когда вам будет предложено отключить некоторые “опциональные” компоненты, не отключайте ни одного! Некоторые из этих компонентов могут вам пригодиться, особенно IDLE.

- **Командная строка**

  Для использования Python из командной строки Windows, необходимо установить должным образом переменную PATH.

  Нажмите кнопку “Пуск” и выберите "Панель Управления" –> "Система и безопасность" –> "Система". Нажмите "Дополнительные параметры системы" слева, а затем выберите вкладку "Дополнительно". Внизу нажмите кнопку "Переменные среды" и в разделе "Системные переменные" найдите переменную PATH, выберите её и нажмите "Редактировать".

Перейдите к концу строки в поле "Значение переменной" и допишите **;C:\путь-куда-установили-python**.
Нажмите “Ok”, и всё. Перезагрузка не требуется.

- **Запуск командной строки Python в Windows**

Если вы должным образом установили значение переменной PATH, теперь можно запускать интерпретатор из командной строки.

Чтобы открыть терминал в Windows, нажмите кнопку "Пуск" и выберите "Выполнить". В появившемся диалоговом окне наберите cmd и нажмите Enter.

Затем наберите **python3 -V** и проверьте, нет ли ошибок.

Использование командной строки интерпретатора

Откройте окно терминала (как было описано в главе Установка) и запустите интерпретатор Python, введя команду python3 и нажав Enter.

Пользователи Windows могут запустить интерпретатор в командной строке, если установили переменную PATH надлежащим образом. Чтобы открыть командную строку в Windows, зайдите в меню "Пуск" и нажмите "Выполнить...". В появившемся диалоговом окне введите "cmd" и нажмите Enter; теперь у вас будет всё необходимое для начала работы с python в командной строке Windows.

Если вы используете IDLE, нажмите "Пуск" –> "Программы" –> "Python 3.0" –> "IDLE (Python GUI)".

Как только вы запустили python3, вы должны увидеть >>> в начале строки, где вы можете что-то набирать. Это и называется командной строкой интерпретатора Python.

Теперь введите print('Hello World') и нажмите клавишу Enter. В результате должна появиться строка "Hello World". Обратите внимание, что Python выдаёт результат работы строки немедленно!

Выйти из командной строки интерпретатора можно нажатием Ctrl-D или введя команду exit() (примечание: не забудьте написать скобки, "()"), а затем нажав клавишу Enter. Если вы используете командную строку Windows, нажмите Ctrl-Z, а затем нажмите клавишу Enter.

Получение помощи

Для быстрого получения информации о любой функции или операторе Python служит встроенная функция **help**. Это особенно удобно при использовании командной строки интерпретатора. К примеру, выполните **help(print)** – это покажет справку по функции print, которая используется для вывода на экран.

Аналогичным образом можно получить информацию почти о чём угодно в Python. При помощи функции help() можно даже получить описание самой функции help!

Если вас интересует информация об операторах, как например, **if**, их необходимо указывать в кавычках (например, **help('if')**), чтобы Python понял, чего мы хотим.

Комментарии

Комментарии – это то, что пишется после символа #, и представляет интерес лишь как заметка для читающего программу.

Например:

```
print('Привет, Мир!) # print -- это функция
```

или:

```
# print -- это функция
print('Привет, Мир!)
```

Старайтесь в своих программах писать как можно больше полезных комментариев, объясняющих:

- предположения;
- важные решения;
- важные детали;
- проблемы, которые вы пытаетесь решить;
- проблемы, которых вы пытаетесь избежать и т.д.

Текст программы говорит о том, КАК, а комментарии должны объяснять, ПОЧЕМУ.

Это будет полезно для тех, кто будет читать вашу программу, так как им легче будет понять, что программа делает. Помните, что таким человеком можете оказаться вы сами через полгода!

Литеральные константы

Примером литеральной константы может быть число, например, 5, 1.23, 9.25e-3 или что-нибудь вроде 'Это строка' или "It's a string!". Они называются литеральными, потому что они "буквальны" – вы используете их значение буквально. Число 2 всегда представляет само себя и ничего другого – это "константа", потому что её значение нельзя изменить.

Поэтому всё это называется литеральными константами.

Числа

Числа в Python бывают трёх типов: целые, с плавающей точкой и комплексные.

Примером целого числа может служить 2.

Примерами чисел с плавающей точкой (или "плавающих" для краткости) могут быть 3.23 и 52.3E-4. Обозначение E показывает степени числа 10. В данном случае 52.3E-4 означает 52.3 * 10-4.

Примеры комплексных чисел: (-5+4j) и (2.3 - 4.6j)

Ниже перечислены операции, которые могут применяться ко всем числовым типам:

- x + y Сложение
- x – y Вычитание
- x * y Умножение
- x / y Деление
- x // y Деление с усечением
- x ** y Возведение в степень
- x % y Деление по модулю
- -x Унарный минус
- +x Унарный плюс

Строки

Строка – это последовательность символов. Чаще всего строки – это просто некоторые наборы слов.

Слова могут быть как на английском языке, так и на любом другом, поддерживаемом стандартом Unicode, что означает почти на любом языке мира.

Вы будете использовать строки почти в каждой вашей программе на Python. Поэтому уделите внимание тому, как работать со строками в Python.

- **Одинарные кавычки**

  Строку можно указать, используя одинарные кавычки, как например, 'Фраза в кавычках'. Все пробелы и знаки табуляции сохранятся, как есть.

- **Двойные кавычки**

  Строки в двойных кавычках работают точно так же, как и в одинарных. Например, "What's your name?".

- **Тройные кавычки**

  Можно указывать "многострочные" строки с использованием тройных кавычек (""" или '''). В пределах тройных кавычек можно свободно использовать одинарные и двойные кавычки. Например:

```
'''Это многострочная строка. Это её первая строка.
Это её вторая строка.
"What's your name?", - спросил я.
Он ответил: "Bond, James Bond."
'''
```

- **Строки неизменяемы**

  Это означает, что после создания строки её больше нельзя изменять. На первый взгляд это может показаться недостатком, но на самом деле это не так.

Впоследствии на примере разных программ мы увидим, почему это не является ограничением.

- **Объединение строковых констант**

Если расположить рядом две строковых константы, Python автоматически их объединит. Например, 'What\'s ' 'your name?' автоматически преобразуется в "What's your name?".

- **Метод format**

Иногда бывает нужно составить строку на основе каких-либо данных. Вот здесь-то и пригодится метод format():

```
age = 25
name = 'John'
print('Имя: {0},  возраст: {1} лет.'.format(name, age))
```

Взгляните на второй случай применения обозначений, где мы пишем {0}, и это соответствует переменной name, являющейся первым аргументом метода format. Аналогично, второе обозначение {1} соответствует переменной age, являющейся вторым аргументом метода format. Заметьте, что Python начинает отсчёт с 0, поэтому первая позиция – номер 0, вторая – номер 1 и т.д.

Заметьте, мы ведь могли добиться того же самого результата и объединением строк:

```
'Имя: ' + name + ', возраст: ' + str(age) + ' лет.'
```

однако вы сами видите, как это некрасиво, и как легко в таком случае допустить ошибку. Во-вторых, преобразование в строку делается методом format автоматически, в отличие от явного преобразования в нашем примере. В-третьих, используя метод format, мы можем изменить сообщение, не затрагивая используемых переменных, и наоборот.

На всякий случай имейте в виду, что цифры здесь не обязательны. Можно было бы просто написать:

```
print('Имя: {},  возраст: {} лет.'.format(name, age))
```

и получить такой же результат, как и ранее.

В методе format Python помещает значение каждого аргумента в обозначенное место. Могут быть и более детальные обозначения, как то:

```
>>> '{0:.3}'.format(1/3) # десятичное число (.) с точностью в 3 знака для плавающих
'0.333'
>>> 'Имя: {name} , возраст: написал {age}'.format(name='John', age=25) # по ключевым словам
'Имя: John , возраст: 25'
```

Начиная с Python 3.6 можно использовать форматированную строку, подставляя имена переменных прямо в фигурные скобки без использования функции format():

```
print(f'Имя: {name},  возраст: {age} лет.')
```

Переменные

Использование одних лишь литеральных констант может скоро наскучить – нам ведь нужен способ хранения любой информации и манипулирования ею. Вот здесь на сцену выходят переменные. Слово "переменные" говорит само за себя – их значение может меняться, а значит, вы можете хранить в переменной всё, что угодно. Переменные – это просто ссылки на области памяти компьютера, в которых вы храните некоторую информацию. В отличие от констант, к такой информации нужно каким-то образом получать доступ, поэтому переменным даются имена.

Имена идентификаторов

Переменные – это частный случай идентификаторов. Идентификаторы – это имена, присвоенные чему-то для его обозначения. При выборе имён для идентификаторов необходимо соблюдать следующие правила:

- Первым символом идентификатора должна быть буква из алфавита (символ ASCII в верхнем или нижнем регистре, или символ Unicode), а также символ подчёркивания ("_").
- Остальная часть идентификатора может состоять из букв (символы ASCII в верхнем или нижнем регистре, а также символы Unicode), знаков подчёркивания ("_") или цифр (0-9).
- Имена идентификаторов чувствительны к регистру. Например, myname и myName – это не одно и то же. Обратите внимание на "n" в нижнем регистре в первом случае и "N" в верхнем во втором.

Примеры допустимых имён идентификаторов: i, __my_name, name_23, a1b2_c3 и любые_символы_utf8_δξ𐌗ђёўЩдΞéά.

Примеры недопустимых имён идентификаторов: 2things, здесь есть пробелы, my-name, >a1b2_c3 и "это_в_кавычках".

Типы данных

Переменные могут хранить значения разных типов, называемых типами данных. Основными типами являются числа и строки, о которых мы уже говорили. На дальнейших занятиях мы познакомимся и с другими типами данных

Объекты

Помните, Python рассматривает всё, что есть в программе, как объекты. Имеется в виду, в самом общем смысле. Вместо того, чтобы говорить "нечто", мы говорим "объект". Python строго объектно ориентирован в том смысле, что объектом является всё, включая числа, строки и функции.

Сейчас мы увидим, как использовать переменные наряду с константами:

```
i = 5
print(i)
i = i + 1
print(i)
s = '''Это многострочная строка.
Это вторая её строчка.'''
print(s)
```

Вот как эта программа работает. Сперва мы присваиваем значение константы 5 переменной i, используя оператор присваивания (=). Эта строка называется предложением и указывает, что должно быть произведено некоторое действие, и в данном случае мы связываем имя переменной i со значением 5. Затем мы печатаем значение i, используя функцию print, которая просто печатает значение переменной на экране.

Далее мы добавляем 1 к значению, хранящемуся в i и сохраняем его там. После этого мы печатаем его и получаем значение 6, что неудивительно.

Аналогичным образом мы присваиваем строковую константу переменной s, после чего печатаем её.

Переменные используются простым присваиванием им значений. Никакого предварительного объявления или определения типа данных не требуется/применяется.

Встроенные функции

Функции – это многократно используемые фрагменты программы. Они позволяют дать имя определённому блоку команд с тем, чтобы впоследствии запускать этот блок по указанному имени в любом месте программы и сколь угодно много раз. Это называется вызовом функции.

Python имеет множество встроенных функций, вот некоторые из них:

abs(x)
Абсолютное значение числа x

round(x, [n])
Округляет число x до ближайшего кратного 10-n (только для чисел с плавающей точкой)

- float(x)
  Преобразует значение x в десятичную дробь
- int(x)
  Преобразует значение x в целое число
- str(x)
  Преобразует значение x в строку
- chr(x)
  Преобразует целое число x в символ
- ord(x)
  Преобразует одиночный символ в целое число.
- hex(x)
  Преобразует целое число x в строку с шестнадцатеричным представлением.
- bin(x)
  Преобразует целое число x в строку с двоичным представлением.
- oct(x)
  Преобразует целое число x в строку с восьмеричным представлением.
- min(x, [y...])
  Возвращает минимальное значение в последовательности чисел
- max(x, [y...])
  Возвращает максимальное значение в последовательности чисел

- len(s)

  Возвращает количество символов в строке s

- print(s, [end='\n'])

  Печатает строку s. Завершающий символ по умолчанию '\n'

- input(s)

  Печатает строку s и ожидает ввода пользователя. Возвращает значение в виде строки

Методы

Хотя тема объектов и классов на данном курсе не рассматривается, всё же необходимо некоторое пояснение, чтобы вы лучше поняли идею объектно-ориентированной парадигмы.

Каждый объект - экземпляр того или иного класса.

Например, строка. Когда мы назначаем некоторой переменной s значение, скажем, "abc", это можно представить себе как создание объекта (т.е. экземпляра) s класса (т.е. типа) str. Чтобы лучше понять это, прочитайте help(str).

Класс может иметь методы, т.е. функции, определённые для использования только применительно к объекту данного класса. Этот функционал будет доступен только когда имеется объект данного класса. Например, Python предоставляет метод upper для класса str, который позволяет получить копию строки (как новый объект), в которой все символы набраны в верхнем регистре. Так "abc".upper() вернёт новую строку в виде "ABC".

Обратите внимание, что для доступа к методу объекта используется конструкция вида "объект.метод()".

Вот список некоторых методов класса str для строковых объектов:

- capitalize()
  Преобразует первый символ в верхний регистр.
- count(sub [,start [,end]])
  Подсчитывает число вхождений заданной подстроки sub.
- endswith(suffix [,start [,end]])
  Проверяет, оканчивается ли строка подстрокой suffix.
- find(sub [, start [,end]])
  Отыскивает первое вхождение подстроки sub или возвращает -1.
- index(sub [, start [,end]])
  Отыскивает первое вхождение подстроки sub или возбуждает исключение.
- isalnum()
  Проверяет, являются ли все символы в строке алфавитно-цифровыми символами.

●isalpha()

Проверяет, являются ли все символы в строке алфавитными символами.

●isdigit()

Проверяет, являются ли все символы в строке цифровыми символами.

●islower()

Проверяет, являются ли все символы в строке символами нижнего регистра.

●isspace()

Проверяет, являются ли все символы в строке пробельными символами.

●istitle()

Проверяет, являются ли первые символы всех слов символами верхнего регистра.

●isupper()

Проверяет, являются ли все символы в строке символами верхнего регистра.

●lower()

Преобразует символы строки в нижний регистр.

●lstrip([chrs])

Удаляет начальные пробельные символы или символы, перечисленные в аргументе chrs.

●replace(old, new [,maxreplace])

Замещает подстроку old подстрокой new.

●rfind(sub [,start [,end]])

Отыскивает последнее вхождение подстроки.

●rindex(sub [,start [,end]])

Отыскивает последнее вхождение подстроки или возбуждает исключение.

●rstrip([chrs])

Удаляет конечные пробельные символы или символы, перечисленные в аргументе chrs.

●startswith(prefix [,start [,end]])

Проверяет, начинается ли строка подстрокой prefix.

●strip([chrs])

Удаляет начальные и конечные пробельные символы или символы, перечисленные в аргументе chrs.

- swapcase()

  Приводит символы верхнего регистра к нижнему, и наоборот.

- title()

  Возвращает версию строки, в которой первые символы всех слов приведены к верхнему регистру.

- upper()

  Преобразует символы строки в верхний регистр.

Не забудьте для помощи набрать в консоли help(str.method), например help(str.upper).

Обратите внимание, что круглые скобки после имени метода указывать не надо.

Логические и физические строки

Физическая строка – это то, что вы видите, когда набираете программу. Логическая строка – это то, что Python видит как единое предложение. Python неявно предполагает, что каждой физической строке соответствует логическая строка.

Примером логической строки может служить предложение print('Привет, Мир!') – если оно на одной строке (как вы видите это в редакторе), то эта строка также соответствует физической строке.

Python неявно стимулирует использование по одному предложению на строку, что облегчает чтение кода.

Чтобы записать более одной логической строки на одной физической строке, вам придётся явно указать это при помощи точки с запятой (;), которая отмечает конец логической строки/предложения. Например:

```
i = 5
print(i)
```

то же самое, что

```
i = 5;
print(i);
```

и то же самое может быть записано в виде

```
i = 5; print(i);
```

или даже

```
i = 5; print(i)
```

Однако я настоятельно рекомендую вам придерживаться написания одной логической строки в каждой физической строке. Таким образом вы можете обойтись совсем без точки с запятой.

Можно использовать более одной физической строки для логической строки, но к этому следует прибегать лишь в случае очень длинных строк. Пример написания одной логической строки, занимающей несколько физических строк, приведён ниже. Это называется явным объединением строк.

```
s = 'Это строка. \
Это строка продолжается.'
print(s)
```

Аналогично:

print\
(i)

то же самое, что и

```
print(i)
```

Иногда имеет место неявное подразумевание, когда использование обратной косой черты не обязательно. Это относится к случаям, когда в логической строке есть открывающаяся круглая, квадратная или фигурная скобка, но нет закрывающейся. Это называется неявным объединением строк. Вы сможете увидеть это в действии в программах с использованием списков на дальнейших занятиях.

Отступы

В Python пробелы важны. Точнее, пробелы в начале строки важны. Это называется отступами. Передние отступы (пробелы и табуляции) в начале логической строки используются для определения уровня отступа логической строки, который, в свою очередь, используется для группировки предложений.

Это означает, что предложения, идущие вместе, должны иметь одинаковый отступ. Каждый такой набор предложений называется блоком. В дальнейших главах мы увидим примеры того, насколько важны блоки.

Вы должны запомнить, что неправильные отступы могут приводить к возникновению ошибок. Например:

```
i = 5
 print('Значение составляет ', i) # Ошибка! Обратите внимание на первый пробел в начале строки
print('Я повторяю, значение составляет ', i)
```

Когда вы запустите это, вы получите следующую ошибку: IndentationError: unexpected indent

Обратите внимание на то, что в начале второй строки есть один пробел. Ошибка, отображённая Python, говорит нам о том, что синтаксис программы неверен, т.е. программа не была написана по правилам. Для вас же это означает, что вы не можете начинать новые блоки предложений где попало (кроме основного блока по умолчанию, который используется на протяжении всей программы, конечно). Случаи, в которых вы можете использовать новые блоки, будут подробно описаны далее.

Как отступать Не смешивайте пробелы и символы табуляции в отступах, поскольку не на всех платформах это работает корректно. Я настоятельно рекомендую вам использовать одиночную табуляцию или четыре пробела для каждого уровня отступа.

Выберите какой-нибудь один из этих стилей отступа. Но что ещё более важно, это использовать выбранный стиль постоянно, а также соблюдать стиль редактируемых вами файлов. Т.е. когда вы пишете новый файл, используйте только один ваш любимый стиль, а

если в редактируемом вами файле для отступов уже используются, скажем, символы табуляции, то и вы используйте в этом файле символы табуляции для отступов. К слову, хорошие редакторы будут делать это автоматически.

Операторы и выражения

Большинство предложений (логических строк) в программах содержат выражения. Простой пример выражения: 2 + 3. Выражение можно разделить на операторы и операнды.

Операторы

Операторы – это некий функционал, производящий какие-либо действия, который может быть представлен в виде символов, как например +, или специальных зарезервированных слов. Операторы могут производить некоторые действия над данными, и эти данные называются операндами. В нашем случае 2 и 3 – это операнды.

Кратко рассмотрим некоторые операторы и их применение:

\+ Сложение
Суммирует два объекта
3 + 5 даст 8, 'a' + 'b' даст 'ab'

\- Вычитание
Даёт разность двух чисел; если первый операнд отсутствует, он считается равным нулю
-5.2 даст отрицательное число, а 50 - 24 даст 26.

* Умножение
Даёт произведение двух чисел или возвращает строку, повторённую заданное число раз. 2 * 3 даст 6, 'la' * 3 даст 'lalala'.

** Возведение в степень
Возвращает число x, возведённое в степень y
3 ** 4 даст 81 (т.е. 3 * 3 * 3 * 3)

/ Деление
Возвращает частное от деления x на y
4 / 3 даст 1.3333333333333333.

// Целочисленное деление

Возвращает неполное частное от деления
4 // 3 даст 1, -4 // 3 даст -2.

% Деление по модулю
Возвращает остаток от деления
8 % 3 даст 2, -25.5 % 2.25 даст 1.5.

Краткая запись мат. операций и присваивания

Зачастую результат проведения некой математической операции необходимо присвоить переменной, над которой эта операция производилась. Для этого существуют краткие формы записи выражений:

Вы можете записать:

```
a = 2
a = a * 3
```

в виде:

```
a = 2;
a *= 3
```

Обратите внимание, что выражения вида "переменная = переменная операция выражение" принимает вид "переменная операция = выражение".

Порядок вычисления

Если имеется выражение вида 2 + 3 * 4, что производится раньше: сложение или умножение? Школьный курс математики говорит нам, что умножение должно производиться в первую очередь. Это означает, что оператор умножения имеет более высокий приоритет, чем оператор сложения.

Изменение порядка вычисления

Для облегчения чтения выражений можно использовать скобки. Например, 2 + (3 * 4) определённо легче понять, чем 2 + 3 * 4, которое требует знания приоритета операторов. Как и всё остальное, скобки нужно использовать разумно (не перестарайтесь) и избегать излишних, как в (2 + (3 * 4)).

Есть ещё одно преимущество в использовании скобок – они дают возможность изменить порядок вычисления выражений. Например, если сложение необходимо произвести прежде умножения, можно записать нечто вроде (2 + 3) * 4.

Ассоциативность

Операторы обычно обрабатываются слева направо. Это означает, что операторы с равным приоритетом будут обработаны по порядку от левого до правого. Например, 2 + 3 + 4 обрабатывается как (2 + 3) + 4.

Выражения

Пример:

```
length = 5
breadth = 2
area = length * breadth
print('Площадь равна', area)
print('Периметр равен', 2 * (length + breadth))
```

Как это работает:

Длина и ширина прямоугольника хранятся в переменных length и breadth соответственно. Мы используем их для вычисления периметра и площади прямоугольника при помощи выражений. Результат выражения length * breadth сохраняется в переменной area, после чего выводится на экран функцией print. Во втором случае мы напрямую подставляем значение выражения 2 * (length + breadth) в функцию print.

Также обратите внимание, как Python “красиво печатает” результат. Несмотря на то, что мы не указали пробела между 'Площадь равна' и переменной area, Python подставляет его за нас, чтобы получить красивый и понятный вывод. Программа же остаётся при этом легкочитаемой (поскольку нам не нужно заботиться о пробелах между строками, которые мы выводим). Это пример того, как Python облегчает жизнь программисту.

Модуль 2

Управляющие конструкции

Булевы значения

Перед тем, как мы поговорим о способах управления нашим кодом, нам нужно познакомиться с ещё одним типом данных. В Python определён класс bool для булевых значений True и False. Обратите внимание на заглавные буквы в этих словах.

Для приведения других типов к булеву используется встроенная функция bool(). Мы пока знаем три типа (не считая самого булева): int, float и str. Вот какие их значения при приведении к булеву типу вернут значение False:

- `bool(0)`
- `bool(0.0)`
- `bool("")`

Приведение других значений к булеву типу вернёт значение True.

Вот список операторов сравнения, которые возвращают значения булева типа.

- < Меньше
  Определяет, верно ли, что x меньше y
  5 < 3 даст False, а 3 < 5 даст True.

- > Больше
  Определяет, верно ли, что x больше y
  5 > 3 даёт True.

  Если оба операнда - числа, то перед сравнением они оба преобразуются к одинаковому типу. В противном случае всегда возвращается False.

- <= Меньше или равно
  Определяет, верно ли, что x меньше или равно y
  x = 3; y = 6; x <= y даёт True.

Больше или равно >=
Определяет, верно ли, что x больше или равно y
x = 4; y = 3; x >= y даёт True.

- == Равно
  Проверяет, одинаковы ли объекты
  x = 2; y = 2; x == y даёт True.

x = 'str'; y = 'stR'; x == y даёт False.

x = 'str'; y = 'str'; x == y даёт True.

- != Не равно

  Проверяет, верно ли, что объекты не равны

  x = 2; y = 3; x != y даёт True.

А это список логических операторов. Они тоже возвращают либо True, либо False.

- not Логическое НЕ

  Если x равно True, оператор вернёт False. Если же x равно False, получим True.

  x = True; not x даёт False.

- and Логическое И

  x and y даёт False, если x равно False , в противном случае возвращает значение y

  x = False; y = True; x and y возвращает False, поскольку x равно False.

  В этом случае Python не станет проверять значение y, так как уже знает, что левая часть выражения 'and' равняется False, что подразумевает, что и всё выражение в целом будет равно False, независимо от значений всех остальных операндов. Это называется укороченной оценкой булевых (логических) выражений.

- or Логическое ИЛИ

  Если x равно True, в результате получим True, в противном случае получим значение y

  x = True; y = False; x or y даёт True.

  Здесь также может производиться укороченная оценка выражений.

Поток команд

В программах, которые мы до сих пор рассматривали, последовательность команд всегда выполнялась Python по порядку строго сверху вниз. А что, если нам необходимо изменить поток выполняющихся команд? Например, если требуется, чтобы программа принимала некоторое решение и выполняла различные действия в зависимости от ситуации; скажем, печатала “Доброе утро” или “Добрый вечер” в зависимости от времени суток.

Как вы уже, наверное, догадались, этого можно достичь при помощи операторов управления потоком.

Оператор if

Оператор if используется для проверки условий: если условие верно, выполняется блок выражений (называемый "if-блок"), иначе выполняется другой блок выражений (называемый "else-блок"). Блок "else" является необязательным.

```
number = 23
guess = int(input('Введите целое число : '))
if guess == number:
    print('Поздравляю, вы угадали,') # Здесь начинается новый блок
print('(хотя и не выиграли никакого приза!)') # Здесь заканчивается
новый блок
elif guess < number:
    print('Нет, загаданное число немного больше этого.') # Ещё один блок
# Внутри блока вы можете выполнять всё, что угодно …
else:
    print('Нет, загаданное число немного меньше этого.')
# чтобы попасть сюда, guess должно быть больше, чем number
print('Завершено')
# Это последнее выражение выполняется всегда после выполнения оператора
if
```

В этой программе мы принимаем варианты от пользователя и проверяем, совпадают ли они с заранее заданным числом. Мы устанавливаем переменной number значение любого целого числа, какого хотим. Например, 23. После этого мы принимаем вариант числа от пользователя при помощи функции input(). Функции – это всего-навсего многократно используемые фрагменты программы. Мы узнаем о них больше в следующем модуле.

Мы передаём встроенной функции input строку, которую она выводит на экран и ожидает ввода от пользователя. Как только мы ввели что-нибудь и нажали клавишу Enter, функция input() возвращает строку, которую мы ввели. Затем мы преобразуем полученную строку в число при помощи int(), и сохраняем это значение в переменную guess.

Далее мы сравниваем число, введённое пользователем, с числом, которое мы выбрали заранее. Если они равны, мы печатаем сообщение об успехе. Обратите внимание, что мы используем соответствующие уровни отступа, чтобы указать Python, какие выражения относятся к какому блоку. Вот почему отступы так важны в Python. Я надеюсь, вы придерживаетесь правила "постоянных отступов", не так ли?

Обратите внимание, что в конце оператора if стоит двоеточие – этим мы показываем, что далее следует блок выражений.

После этого мы проверяем, верно ли, что пользовательский вариант числа меньше загаданного, и если это так, мы информируем пользователя о том, что ему следует выбирать числа немного больше этого. Здесь мы использовали выражение elif, которое попросту объединяет в себе два связанных if else-if else выражения в одно выражение if-elif-else. Это облегчает чтение программы, а также не требует дополнительных отступов.

Выражения elif и else также имеют двоеточие в конце логической строки, за которым следуют соответствующие блоки команд (с соответствующим числом отступов, конечно).

Внутри if-блока оператора if может быть другой оператор if и так далее – это называется вложенным оператором if.

Помните, что части elif и else не обязательны. Минимальная корректная запись оператора if такова:

```
if True:
    print('Да, это верно.')
```

После того, как Python заканчивает выполнение всего оператора if вместе с его частями elif и else, он переходит к следующему выражению в блоке, содержащем этот оператор if. В нашем случае это основной блок программы (в котором начинается выполнение программы), а следующее выражение – это print('Завершено'). После этого Python доходит до конца программы и просто выходит из неё.

Хотя это и чрезвычайно простая программа, я указал вам на целый ряд вещей, которые стоит взять на заметку. Всё это довольно легко. Поначалу вам придётся держать все эти

вещи в памяти, но после некоторой практики вы привыкнете, и они вам покажутся вполне "естественными".

Оператор while

Оператор while позволяет многократно выполнять блок команд до тех пор, пока выполняется некоторое условие. Это один из так называемых операторов цикла. Он также может иметь необязательный пункт else.

```
number = 23
running = True
while running:
    guess = int(input('Введите целое число : '))
    if guess == number:
    print('Поздравляю, вы угадали.')
    running = False # это останавливает цикл while      elif
guess < number:
    print('Нет, загаданное число немного больше этого')
else:
    print('Нет, загаданное число немного меньше этого.')
else:
    print('Цикл while закончен.')
    # Здесь можете выполнить всё что вам ещё нужно
print('Завершение.')
```

В этой программе мы продолжаем играть в игру с угадыванием, но преимущество состоит в том, что теперь пользователь может угадывать до тех пор, пока не угадает правильное число, и ему не придётся запускать программу заново для каждой попытки, как это происходило до сих пор. Это наглядно демонстрирует применение оператора while.

Мы переместили операторы input и if внутрь цикла while и установили переменную running в значение True перед запуском цикла. Прежде всего проверяется, равно ли значение переменной running True, а затем происходит переход к соответствующему while-блоку. После выполнения этого блока команд условие, которым в данном случае является переменная running, проверяется снова. Если оно истинно, while-блок запускается снова, в противном случае происходит переход к дополнительному else-блоку, а затем – к следующему оператору.

Блок else выполняется тогда, когда условие цикла while становится ложным (False) – это может случиться даже при самой первой проверке условия. Если у цикла while имеется дополнительный блок else, он всегда выполняется, если только цикл не будет прерван оператором break.

True и False называются булевым типом данных, и вы можете считать их эквивалентными значениям 1 и 0 соответственно.

Оператор break

Оператор break служит для прерывания цикла, т.е. остановки выполнения команд даже если условие выполнения цикла ещё не приняло значения False или последовательность элементов не закончилась.

Важно отметить, что если цикл прервать оператором break, соответствующие им блоки else выполняться не будут.

```
while True:
    s = input('Введите что-нибудь : ')
    if s == 'выход':
    Break
    print('Длина строки:', len(s))
print('Завершение')
```

В этой программе мы многократно считываем пользовательский ввод и выводим на экран длину каждой введённой строки. Для остановки программы мы вводим специальное условие, проверяющее, совпадает ли пользовательский ввод со строкой 'выход'. Мы останавливаем программу прерыванием цикла оператором break и достигаем её конца.

Длина введённой строки может быть найдена при помощи встроенной функции len.

Оператор continue

Оператор continue используется для указания Python, что необходимо пропустить все оставшиеся команды в текущем блоке цикла и продолжить со следующей итерации цикла.

```
while True:
    s = input('Введите что-нибудь : ')
    if s == 'выход':
    Break
    if len(s) < 3:
    print('Слишком мало')
    Continue
    print('Введённая строка достаточной длины')
    # Разные другие действия здесь...
```

В этой программе мы запрашиваем ввод со стороны пользователя, но обрабатываем введённую строку только если она имеет длину хотя бы в 3 символа. Итак, мы используем встроенную функцию len для получения длины строки, и если длина менее 3, мы пропускаем остальные действия в блоке при помощи оператора continue. В противном случае все остальные команды в цикле выполняются, производя любые манипуляции, которые нам нужны.

Модуль 3

Функции и коллекции

Цикл for

Оператор for..in является оператором цикла, который осуществляет итерацию по последовательности объектов, т.е. проходит через каждый элемент в последовательности. Последовательность – это упорядоченный набор элементов.

```
for i in range(1, 5):
    print(i)
else:
    print('Цикл for закончен')
```

В этой программе мы выводим на экран последовательность чисел. Мы генерируем эту последовательность, используя встроенную функцию range.

Мы задаём два числа, и range возвращает последовательность чисел от первого числа до второго. Например, range(1, 5) даёт последовательность [1, 2, 3, 4]. По умолчанию range принимает значение шага, равное 1. Если мы зададим также и третье число range, оно будет служить шагом. Например, range(1, 5, 2) даст [1, 3]. Помните, интервал простирается только до второго числа, т.е. не включает его в себя.

Обратите внимание, что range() генерирует последовательность чисел, но только по одному числу за раз – когда оператор for запрашивает следующий элемент. Чтобы увидеть всю последовательность чисел сразу, используйте list(range()).

Затем цикл for осуществляет итерацию по этому диапазону - for i in range(1, 5) эквивалентно for i in [1, 2, 3, 4], что напоминает присваивание переменной i по одному числу (или объекту) за раз, выполняя блок команд для каждого значения i. В данном случае в блоке команд мы просто выводим значение на экран.

Помните, что блок else не обязателен. Если он присутствует, он всегда выполняется один раз после окончания цикла for, если только не указан оператор break.

Помните также, что цикл for..in работает для любой последовательности. В нашем случае это список чисел, сгенерированный встроенной функцией range, но в общем случае можно использовать любую последовательность любых объектов! Далее мы познакомимся с этим поближе.

Структуры данных

Структуры данных – это, по сути, и есть структуры, которые могут хранить некоторые данные вместе. Другими словами, они используются для хранения связанных данных.

В Python существуют четыре встроенных структуры данных: список, кортеж, словарь и множество. Посмотрим, как ими пользоваться, и как они могут облегчить нам жизнь.

Список

Список – это структура данных, которая содержит упорядоченный набор элементов, т.е. хранит последовательность элементов. Это легко представить, если вспомнить список покупок, в котором перечисляется, что нужно купить, с тем лишь исключением, что в списке покупок каждый элемент обычно размещается на отдельной строке, тогда как в Python они разделяются запятыми.

Список элементов должен быть заключён в квадратные скобки, чтобы Python понял, что это список. Как только список создан, можно добавлять, удалять или искать элементы в нём. Поскольку элементы можно добавлять и удалять, мы говорим, что список – это изменяемый тип данных, т.е. его можно модифицировать.

Список – это один из примеров использования объектов и классов. Когда мы назначаем некоторой переменной i значение, скажем, целое число 5, это можно представить себе как создание объекта (т.е. экземпляра) i класса (т.е. типа) int. Чтобы лучше понять это, прочитайте help(int).

Класс может также иметь методы, т.е. функции, определённые для использования только применительно к данному классу. Этот функционал будет доступен только когда имеется объект данного класса. Например, Python предоставляет метод append для класса list, который позволяет добавлять элемент к концу списка. Так mylist.append('and item') добавит эту строку к списку mylist. Обратите внимание на обозначение точкой для доступа к методам объектов.

Класс также может иметь поля, которые представляют собой не что иное, как переменные, определённые для использования только применительно к данному классу. Эти переменные/имена можно использовать только тогда, когда имеется объект этого класса. Доступ к полям также осуществляется при помощи точки. Например, mylist.field.

```
# Это мой список покупок
shoplist = ['яблоки', 'манго', 'морковь', 'бананы']
print('Я должен сделать', len(shoplist), 'покупки.')
print('Покупки:', end=' ')
for item in shoplist:
    print(item, end=' ')
print('\nТакже нужно купить риса.')
shoplist.append('рис')
print('Теперь мой список покупок таков:', shoplist)
print('Отсортирую-ка я свой список')
shoplist.sort()
print('Отсортированный список покупок выглядит так:', shoplist)
print('Первое, что мне нужно купить, это', shoplist[0])
olditem = shoplist[0]
del shoplist[0]
print('Я купил', olditem)
print('Теперь мой список покупок:', shoplist)
```

Переменная shoplist – это список покупок человека, идущего на рынок. В shoplist мы храним только строки с названиями того, что нужно купить, однако в список можно добавлять любые объекты, включая числа или даже другие списки.

Мы также использовали цикл for..in для итерации по элементам списка. Вы уже, наверное, поняли, что список является также и последовательностью. Особенности последовательностей будут рассмотрены ниже.

Обратите внимание на использование ключевого аргумента end в функции print, который показывает, что мы хотим закончить вывод пробелом вместо обычного перевода строки.

Далее мы добавляем элемент к списку при помощи append – метода объекта списка, который уже обсуждался ранее. Затем мы проверяем, действительно ли элемент был добавлен к списку, выводя содержимое списка на экран при помощи простой передачи этого списка функции print, которая аккуратно его печатает.

Затем мы сортируем список, используя метод sort объекта списка. Имейте в виду, что этот метод действует на сам список, а не возвращает изменённую его версию. В этом отличие от того, как происходит работа со строками. Именно это имеется в виду, когда мы говорим, что списки изменяемы, а строки – неизменяемы.

Далее после совершения покупки мы хотим удалить её из списка. Это достигается применением оператора del. Мы указываем, какой элемент списка мы хотим удалить, и оператор del удаляет его. Мы указываем, что хотим удалить первый элемент списка, и поэтому пишем "del shoplist[0]" (помните, что Python начинает отсчёт с 0).

Чтобы узнать более детально обо всех методах объекта списка, просмотрите help(list).

Методысписков

- list(s)
  Преобразует объект s в список.
- l.append(x)
  Добавляет новый элемент x в конец списка l.
- l.extend(t)
  Добавляет новый список t в конец списка l.
- l.count(x)
  Определяет количество вхождений x в список l.
- l.index(x [,start [,stop]])
  Возвращает наименьшее значение индекса i, где s[i] ==
  x. Необязательные значения start и stop определяют индексы начального и конеч ого элементов диапазона, где выполняется поиск.
- l.insert(i,x)
  Вставляет x в элемент с индексом i.
- l.pop([i])
  Возвращает i-й элемент и удаляет его из списка. Если индекс i не указан, возвращается последний элемент.
- l.remove(x)
  Отыскивает в списке s элемент со значением x и удаляет его.
- l.reverse()
  Изменяет порядок следования элементов в списке l на обратный. l.sort([key [, reverse]])
  Сортирует элементы списка l. key – это функция, которая вычисляет значение ключа. reverse – признак сортировки в обратном порядке. Аргументы key и reverse всегда должны передаваться как именованные аргументы.

Кортеж

Кортежи служат для хранения нескольких объектов вместе. Их можно рассматривать как аналог списков, но без такой обширной функциональности, которую предоставляет класс списка. Одна из важнейших особенностей кортежей заключается в том, что они неизменяемы, так же, как и строки. Т.е. модифицировать кортежи невозможно.

Кортежи обозначаются указанием элементов, разделённых запятыми; по желанию их можно ещё заключить в круглые скобки.

Кортежи обычно используются в тех случаях, когда оператор или пользовательская функция должны наверняка знать, что набор значений, т.е. кортеж значений, не изменится.

```
zoo = ('питон', 'слон', 'пингвин') # помните, что скобки не обязательны
print('Количество животных в зоопарке -', len(zoo))
new_zoo = 'обезьяна', 'верблюд', zoo
print('Количество клеток в зоопарке -', len(new_zoo))
print('Все животные в новом зоопарке:', new_zoo)
print('Животные, привезённые из старого зоопарка:', new_zoo[2])
print('Последнее животное, привезённое из старого зоопарка -',
new_zoo[2][2])
print('Количество животных в новом зоопарке -', len(new_zoo)-
1+len(new_zoo[2]))
```

Переменная zoo обозначает кортеж элементов. Как мы видим, функция len позволяет получить длину кортежа. Это также указывает на то, что кортеж является последовательностью.

Теперь мы перемещаем этих животных в новый зоопарк, поскольку старый зоопарк закрывается. Поэтому кортеж new_zoo содержит тех животных, которые уже там, наряду с привезёнными из старого зоопарка. Возвращаясь к реальности, обратите внимание на то, что кортеж внутри кортежа не теряет своей индивидуальности.

Доступ к элементам кортежа осуществляется указанием позиции элемента, заключённой в квадратные скобки – точно так же, как мы это делали для списков. Это называется оператором индексирования. Доступ к третьему элементу в new_zoo мы получаем, указывая new_zoo[2], а доступ к третьему элементу внутри третьего элемента в кортеже new_zoo – указывая new_zoo[2][2]. Это достаточно просто, как только вы поймёте принцип.

Скобки

Хотя скобки и не являются обязательными, лучше их указывать, чтобы было очевидно, что это кортеж, особенно в двусмысленных случаях. Например, print(1, 2, 3) и print((1, 2, 3)) делают разные вещи: первое выражение выводит три числа, тогда как второе – кортеж, содержащий эти три числа.

Кортеж, содержащий 0 или 1 элемент Пустой кортеж создаётся при помощи пустой пары скобок – "myempty = ()". Однако, с кортежем из одного элемента не всё так просто. Его нужно указывать при помощи запятой после первого (и единственного) элемента, чтобы Python мог отличить кортеж от скобок, окружающих объект в выражении. Таким образом, чтобы получить кортеж, содержащий элемент 2, вам потребуется указать "singleton = (2,)".

Словарь

Словарь – это некий аналог адресной книги, в которой можно найти адрес или контактную информацию о человеке, зная лишь его имя; т.е. некоторые ключи (имена) связаны со значениями (информацией). Заметьте, что ключ должен быть уникальным – вы ведь не сможете получить корректную информацию, если у вас записаны два человека с полностью одинаковыми именами.

Обратите также внимание на то, что в словарях в качестве ключей могут использоваться только неизменяемые объекты (как строки), а в качестве значений можно использовать как неизменяемые, так и изменяемые объекты. Точнее говоря, в качестве ключей должны использоваться только простые объекты.

Пары ключ-значение указываются в словаре следующим образом: "d = {key1 : value1, key2 : value2 }". Обратите внимание, что ключ и значение разделяются двоеточием, а пары друг от друга отделяются запятыми, а затем всё это заключается в фигурные скобки.

Помните, что пары ключ-значение никоим образом не упорядочены в словаре. Если вам необходим некоторый порядок, вам придётся отдельно отсортировать словарь перед обращением к нему.

Словари являются экземплярами/объектами класса dict.

```
ab = {
    'Lerdorf'   : 'rasmus@lerdorf.com',
    'Larry'     : 'larry@wall.org',
    'Matsumoto' : 'matz@ruby-lang.org',
    'Spammer'   : 'spammer@hotmail.com'
}
print("Адрес Lerdorfa'a:", ab['Lerdorf'])
# Удаление пары ключ-значение
del ab['Spammer']
print('\nB адресной книге {0} контакта\n'.format(len(ab)))
for name, address in ab.items():
    print('Контакт {0} с адресом {1}'.format(name, address))
# Добавление пары ключ-значение
ab['Guido'] = 'guido@python.org'
if 'Guido' in ab:
    print("\nАдрес Guido:", ab['Guido'])
```

Мы создаём словарь ab при помощи обозначений, описанных ранее. Затем мы обращаемся к парам ключ-значение, указывая ключ в операторе индексирования, которым мы пользовались для списков и кортежей. Как видите, синтаксис прост.

Удалять пары ключ-значение можно при помощи нашего старого доброго оператора del. Мы просто указываем имя словаря и оператор индексирования для удаляемого ключа, после чего передаём это оператору del. Для этой операции нет необходимости знать, какое значение соответствует данному ключу.

Далее мы обращаемся ко всем парам ключ-значение нашего словаря, используя метод items, который возвращает список кортежей, каждый из которых содержит пару элементов: ключ и значение. Мы получаем эту пару и присваиваем её значение переменным name и address соответственно в цикле for..in, а затем выводим эти значения на экран в блоке for.

Новые пары ключ-значение добавляются простым обращением к нужному ключу при помощи оператора индексирования и присваиванием ему некоторого значения, как мы сделали для Guido в примере выше.

Проверить, существует ли пара ключ-значение, можно при помощи оператора in.

Чтобы просмотреть список всех методов класса dict смотрите help(dict).

Методыиоператоры,поддерживаемыесловарями

- ●len(d)
  Возвращает количество элементов в словаре d.
- ●d[k]
  Возвращает элемент словаря d с ключом k.
- ●d[k]=x
  Записывает в элемент d[k] значение x.
- ●del d[k]
  Удаляет элемент d[k].
- ●k in d
  Возвращает True, если ключ k присутствует в словаре d.
- ●d.clear()
  Удаляет все элементы из словаря d.
- ●d.copy()
  Создает копию словаря d.
- ●d.fromkeys(s [,value])
  Создает новый словарь с ключами, перечисленными в последовательности s, а все значения устанавливает равными value.
- ●d.get(k [,v])
  Возвращает элемент d[k], если таковой имеется, в противном случае возвращает v.
- ●d.items()
  Возвращает последовательность пар (key, value).
- ●d.keys()
  Возвращает последовательность ключей.
- ●d.pop(k [,default])
  Возвращает элемент d[k], если таковой имеется, и удаляет его из словаря; в противном случае возвращает default, если этот аргумент указан, или возбуждает исключение KeyError.
- ●d.popitem()
  Удаляет из словаря случайную пару (key, value) и возвращает ее в виде кортежа.

- d.setdefault(k [, v])

  Возвращает элемент d[k], если таковой имеется, в противном случае возвращает значение v и создает новый элемент словаря d[k] = v.

- d.update(b)

  Добавляет все объекты из b в словарь d.

- d.values()

  Возвращает последовательность всех значений в словаре d.

Последовательности

Списки, кортежи и строки являются примерами последовательностей. Но что такое последовательности и что в них такого особенного?

Основные возможности – это проверка принадлежности (т.е. выражения "in" и "not in") и оператор индексирования, позволяющий получить напрямую некоторый элемент последовательности.

Все три типа последовательностей, упоминавшиеся выше (списки, кортежи и строки), также предоставляют операцию получения вырезки, которая позволяет получить вырезку последовательности, т.е. её фрагмент.

```
shoplist = ['яблоки', 'манго', 'морковь', 'бананы']
name = 'swaroop'
# Операция индексирования
print('Элемент 0 -', shoplist[0])
print('Элемент 1 -', shoplist[1])
print('Элемент 2 -', shoplist[2])
print('Элемент 3 -', shoplist[3])
print('Элемент -1 -', shoplist[-1])
print('Элемент -2 -', shoplist[-2])
print('Символ 0 -', name[0])
# Вырезка из списка
print('Элементы с 1 по 3:', shoplist[1:3])
print('Элементы с 2 до конца:', shoplist[2:])
print('Элементы с 1 по -1:', shoplist[1:-1])
print('Элементы от начала до конца:', shoplist[:])
# Вырезка из строки
print('Символы с 1 по 3:', name[1:3])
print('Символы с 2 до конца:', name[2:])
print('Символы с 1 до -1:', name[1:-1])
print('Символы от начала до конца:', name[:])
```

Прежде всего, мы видим, как использовать индексы для получения отдельных элементов последовательности. Это ещё называют приписыванием индекса. Когда мы указываем число в квадратных скобках после последовательности, как показано выше, Python извлекает элемент, соответствующий указанной позиции в последовательности. Помните, что Python начинает отсчёт с 0. Поэтому shoplist[0] извлекает первый элемент, а shoplist[3] – четвёртый элемент последовательности shoplist.

Индекс также может быть отрицательным числом. В этом случае позиция отсчитывается от конца последовательности. Поэтому shoplist[-1] указывает на последний элемент последовательности shoplist, а shoplist[-2] – на предпоследний.

Операция вырезки производится при помощи указания имени последовательности, за которым может следовать пара чисел, разделённых двоеточием и заключённых в

квадратные скобки. Заметьте, как это похоже на операцию индексирования, которой мы пользовались до сих пор. Помните, что числа в скобках необязательны, тогда как двоеточие – обязательно.

Первое число (перед двоеточием) в операции вырезки указывает позицию, с которой вырезка должна начинаться, а второе число (после двоеточия) указывает, где вырезка должна закончиться. Если первое число не указано, Python начнёт вырезку с начала последовательности. Если пропущено второе число, Python закончит вырезку у конца последовательности. Обратите внимание, что полученная вырезка будет начинаться с указанной начальной позиции, а заканчиваться прямо перед указанной конечной позицией, т.е. начальная позиция будет включена в вырезку, а конечная – нет.

Таким образом, shoplist[1:3] возвращает вырезку из последовательности, начинающуюся с позиции 1, включает позицию 2, но останавливается на позиции 3, и поэтому возвращает вырезку из двух элементов. Аналогично, shoplist[:] возвращает копию всей последовательности.

Вырезка может осуществляться и с отрицательными значениями. Отрицательные числа обозначают позицию с конца последовательности. Например, shoplist[:-1] вернёт вырезку из последовательности, исключающую последний элемент, но содержащую все остальные.

Кроме того, можно также указать третий аргумент для вырезки, который будет обозначать шаг вырезки (по умолчанию шаг вырезки равен 1):

```
>>> shoplist = ['яблоки', 'манго', 'морковь', 'бананы']
>>> shoplist[::1]
['яблоки', 'манго', 'морковь', 'бананы']
>>> shoplist[::2]
['яблоки', 'морковь']
>>> shoplist[::3]
['яблоки', 'бананы']
>>> shoplist[::-1]
['бананы', 'морковь', 'манго', 'яблоки']
```

Обратите внимание на то, что когда шаг равен 2, мы получаем элементы, находящиеся на позициях 0, 2, ... Когда шаг равен 3, мы получаем элементы с позиций 0, 3, ... и т.д.

Попробуйте разные комбинации параметров вырезки, используя интерактивную оболочку интерпретатора Python, т.е. его командную строку, чтобы сразу видеть результат. Последовательности замечательны тем, что они дают возможность обращаться к кортежам, спискам и строкам одним и тем же способом!

Операции над последовательностями

- s + r

  Конкатенация

- s * n

  Создает n копий последовательности s, где n – целое число

- all(s)

  Возвращает True, если все элементы последовательности s оцениваются, как истинные

- any(s)

  Возвращает True, если хотя бы один элемент последовательности s оценивается, как истинный

- len(s)

  Длина последовательности

- min(s)

  Минимальный элемент в последовательности s

- max(s)

  Максимальный элемент в последовательности s

- sum(s [, initial])

  Сумма элементов последовательности с необязательным начальным значением

Множество

Множества – это неупорядоченные наборы простых объектов. Они необходимы тогда, когда присутствие объекта в наборе важнее порядка или того, сколько раз данный объект там встречается.

Используя множества, можно осуществлять проверку принадлежности, определять, является ли данное множество подмножеством другого множества, находить пересечения множеств и так далее.

```
>>> bri = set(['Бразилия', 'Россия', 'Индия'])
>>> 'Индия' in bri
True
>>> 'США' in bri
False
>>> bric = bri.copy()
>>> bric.add('Китай')
>>> bric.issuperset(bri)
True
>>> bri.remove('Россия')
>>> bri & bric # OR bri.intersection(bric)
{'Бразилия', 'Индия'}
```

Этот пример достаточно нагляден, так как использует основы теории множеств из школьного курса математики.

Методыиоператоры,поддерживаемыемножествами

- len(s)
  Возвращает количество элементов в множестве s.
- s.copy()
  Создает копию множества s.
- s.difference(t)
  Разность множеств. Возвращает все элементы из множества s, отсутствующие в t.
- s.intersection(t)
  Пересечение множеств. Возвращает все элементы, присутствующие в обоих множествах s и t.
- s.isdisjoint(t)
  Возвращает True, если множества s и t не имеют общих элементов.
- s.issubset(t)
  Возвращает True, если множество s является подмножеством t.
- s.issuperset(t)
  Возвращает True, если множество s является надмножеством t.
- s.symmetric_difference(t)
  Симметричная разность множеств. Возвращает все элементы, которые присутствуют в множестве s или t, но не в обоих сразу.
- s.union(t)
  Объединение множеств. Возвращает все элементы, присутствующие в множестве s или t.
- s.add(item)
  Добавляет элемент item в s. Ничего не делает, если этот элемент уже имеется в множестве.
- s.clear()
  Удаляет все элементы из множества s.
- s.difference_update(t)
  Удаляет все элементы из множества s, которые присутствуют в t.

- s.discard(item)

  Удаляет элемент item из множества s. Ничего не делает, если этот элемент отсутствует в множестве.

- s.intersection_update(t)

  Находит пересечение s и t и оставляет результат в s.

- s.pop()

  Возвращает произвольный элемент множества и удаляет его из s.

Ссылки

Когда мы создаём объект и присваиваем его переменной, переменная только ссылается на объект, а не представляет собой этот объект! То есть имя переменной указывает на ту часть памяти компьютера, где хранится объект. Это называется привязкой имени к объекту.

Обычно вам не следует об этом беспокоиться, однако есть некоторый неочевидный эффект, о котором нужно помнить:

```
print('Простое присваивание')
shoplist = ['яблоки', 'манго', 'морковь', 'бананы']
mylist = shoplist # mylist - лишь ещё одно имя, указывающее на тот же объект!

del shoplist[0] # Я сделал первую покупку, поэтому удаляю её из списка
print('shoplist:', shoplist)
print('mylist:', mylist)
# Обратите внимание, что и shoplist, и mylist выводят один и тот же список
# без пункта "яблоко", подтверждая тем самым, что они указывают на один объект.

print('Копирование при помощи полной вырезки')
mylist = shoplist[:] # создаём копию путём полной вырезки
del mylist[0] # удаляем первый элемент
print('shoplist:', shoplist)
print('mylist:', mylist)
# Обратите внимание, что теперь списки разные
```

Фактически всё объяснение кода содержится в комментариях.

Помните, что если вам нужно сделать копию списка или подобной последовательности, или другого сложного объекта (не такого простого объекта, как целое число), вам следует воспользоваться операцией вырезки. Если вы просто присвоите имя переменной другому имени, оба они будут ссылаться на один и тот же объект, а это может привести к проблемам, если вы не осторожны.

Ещё о строках

Мы уже детально обсуждали строки ранее. Что же ещё можно о них узнать? Что ж, вы знали, например, что строки также являются объектами и имеют методы, при помощи которых можно делать практически всё: от проверки части строки до удаления краевых пробелов?

Все строки, используемые вами в программах, являются объектами класса str. Некоторые полезные методы этого класса продемонстрированы на примере ниже. Чтобы посмотреть весь список методов, выполните help(str).

```
name = 'Popandopulo' # Это объект строки
if name.startswith('Pop'):
    print('Да, строка начинается на "Pop"')
if 'a' in name:
    print('Да, она содержит строку "a"')
if name.find('dop') != -1:
    print('Да, она содержит строку "dop"')
delimiter = '_*_'
mylist = ['Бразилия', 'Россия', 'Индия', 'Китай']
print(delimiter.join(mylist))
```

Здесь мы видим сразу несколько методов строк в действии. Метод startswith служит для того, чтобы определять, начинается ли строка с некоторой заданной подстроки. Оператор in используется для проверки, является ли некоторая строка частью данной строки.

Метод find используется для определения позиции данной подстроки в строке; find возвращает -1, если подстрока не обнаружена. В классе str также имеется отличный метод для объединения (join) элементов последовательности с указанной строкой в качестве разделителя между элементами, возвращающий большую строку, сгенерированную таким образом.

Генераторы списков

Генераторы списков служат для создания новых списков на основе существующих.

Представьте, что имеется список чисел, на основе которого требуется получить новый список, состоящий из всех чисел, умноженных на 2, но только при условии, что само число больше 2. Генераторы списков подходят для таких задач как нельзя лучше.

```
listone = [2, 3, 4]
listtwo = [2*i for i in listone if i > 2]

print(listtwo)
```

В этом примере мы создаём новый список, указав операцию, которую необходимо произвести (2 * i), когда выполняется некоторое условие (if i > 2). Обратите внимание, что исходный список при этом не изменяется.

Преимущество использования генераторов списков состоит в том, что это заметно сокращает объёмы стандартного кода, необходимого для циклической обработки каждого элемента списка и сохранения его в новом списке.

Итераторы

Когда мы создаём список, то можем считывать его элементы один за другим — это называется итерацией:

```
>>> mylist = [1, 2, 3]
>>> for i in mylist :
...    print(i)
```

mylist является итерируемым объектом. Когда мы создаём список, используя генераторное выражение, мы также создаём итератор:

```
>>> mylist = [x*x for x in range(3)]
>>> for i in mylist :
...    print(i)
```

Всё, к чему можно применить конструкцию "for in", является итерируемым объектом: списки, строки, файлы и т.д. Это удобно, потому что можно считывать из них значения сколько потребуется — однако все значения хранятся в памяти, а это не всегда желательно, если у нас много значений.

Генераторы

Генераторы это тоже итерируемые объекты, но прочитать их можно лишь один раз. Это связано с тем, что они не хранят значения в памяти, а генерируют их на лету:

```
>>> mygenerator = (x*x for x in range(3))
>>> for i in mygenerator :
...    print(i)
```

Всё то же самое, разве что используются круглые скобки вместо квадратных. Но нельзя применить конструкцию "for in" второй раз, так как генератор может быть использован только единожды: он вычисляет 0, потом забывает про него и вычисляет 1, завершая вычислением 4 — одно за другим.

Функции

Функции – это многократно используемые фрагменты программы. Они позволяют дать имя определённому блоку команд с тем, чтобы впоследствии запускать этот блок по указанному имени в любом месте программы и сколь угодно много раз. Это называется вызовом функции. Мы уже использовали много встроенных функций, как то len и range.

Функция – это, пожалуй, наиболее важный строительный блок любой нетривиальной программы (на любом языке программирования), поэтому в этой главе мы рассмотрим различные аспекты функций.

Функции определяются при помощи зарезервированного слова def. После этого слова указывается имя функции, за которым следует пара скобок, в которых можно указать имена некоторых переменных, и заключительное двоеточие в конце строки. Далее следует блок команд, составляющих функцию. На примере можно видеть, что на самом деле это очень просто:

```
def sayHello():
    print('Привет, Мир!') # блок, принадлежащий функции
# Конец функции
sayHello() # вызов функции
```

Мы определили функцию с именем sayHello, используя описанный выше синтаксис. Эта функция не принимает параметров, поэтому в скобках не объявлены какие-либо переменные. Параметры функции – это некие входные данные, которые мы можем передать функции, чтобы получить соответствующий им результат.

Обратите внимание, что мы можем вызывать одну и ту же функцию много раз, а значит нет необходимости писать один и тот же код снова и снова.

Параметры функций

Функции могут принимать параметры, т.е. некоторые значения, передаваемые функции для того, чтобы она что-либо сделала с ними. Эти параметры похожи на переменные, за исключением того, что значение этих переменных указывается при вызове функции, и во время работы функции им уже присвоены их значения.

Параметры указываются в скобках при объявлении функции и разделяются запятыми.

Аналогично мы передаём значения, когда вызываем функцию. Обратите внимание на терминологию: имена, указанные в объявлении функции, называются параметрами, тогда как значения, которые вы передаёте в функцию при её вызове, – аргументами.

```
def printMax(a, b):
    if a > b:
    print(a, 'максимально')
    elif a == b:
    print(a, 'равно', b)
    else:
    print(b, 'максимально')


printMax(3, 4) # прямая передача значений
x = 5
y = 7
printMax(x, y) # передача переменных в качестве аргументов
```

Здесь мы определили функцию с именем printMax, которая использует два параметра с именами a и b. Мы находим наибольшее число с применением простого оператора if..else и выводим это число.

При первом вызове функции printMax мы напрямую передаём числа в качестве аргументов. Во втором случае мы вызываем функцию с переменными в качестве аргументов. printMax(x, y) назначает значение аргумента x параметру a, а значение аргумента y – параметру b. В обоих случаях функция printMax работает одинаково.

Локальные переменные

При объявлении переменных внутри определения функции, они никоим образом не связаны с другими переменными с таким же именем за пределами функции – т.е. имена переменных являются локальными в функции. Это называется областью видимости переменной. Область видимости всех переменных ограничена блоком, в котором они объявлены, начиная с точки объявления имени.

```
x = 50

def func(x):
    print('x равен', x)
    x = 2
    print('Замена локального x на', x)

func(x)
print('x по-прежнему', x)
```

При первом выводе значения, присвоенного имени x, в первой строке функции Python использует значение параметра, объявленного в основном блоке, выше определения функции.

Далее мы назначаем x значение 2. Имя x локально для нашей функции. Поэтому когда мы заменяем значение x в функции, x, объявленный в основном блоке, остаётся незатронутым.

Последним вызовом функции print мы выводим значение x, указанное в основном блоке, подтверждая таким образом, что оно не изменилось при локальном присваивании значения в ранее вызванной функции.

Зарезервированное слово "global"

Чтобы присвоить некоторое значение переменной, определённой на высшем уровне программы (т.е. не в какой-либо области видимости, как то функции или классы), необходимо указать Python, что её имя не локально, а глобально (global). Сделаем это при помощи зарезервированного слова global. Без применения зарезервированного слова global невозможно присвоить значение переменной, определённой за пределами функции.

Можно использовать уже существующие значения переменных, определённых за пределами функции (при условии, что внутри функции не было объявлено переменной с таким же именем). Однако, это не приветствуется, и его следует избегать, поскольку человеку, читающему текст программы, будет непонятно, где находится объявление переменной. Использование зарезервированного слова global достаточно ясно показывает, что переменная объявлена в самом внешнем блоке.

```
x = 50

def func(x):
    global x
    print('x равен', x)
    x = 2
    print('Замена глобальное значение x на', x)

func()
print('Значение x составляет', x)
```

Зарезервированное слово global используется для того, чтобы объявить, что x – это глобальная переменная, а значит, когда мы присваиваем значение имени x внутри функции, это изменение отразится на значении переменной x в основном блоке программы.

Используя одно зарезервированное слово global, можно объявить сразу несколько переменных: global x, y, z.

Зарезервированное слово "nonlocal"

Мы увидели, как получать доступ к переменным в локальной и глобальной области видимости. Есть ещё один тип области видимости, называемый “нелокальной” (nonlocal) областью видимости, который представляет собой нечто среднее между первыми двумя.

Нелокальные области видимости встречаются, когда вы определяете функции внутри функций.

Поскольку в Python всё является выполнимым кодом, вы можете определять функции где угодно. Давайте рассмотрим пример:

```
def func_outer():
    x = 2
    print('x равно', x)

    def func_inner():
        nonlocal x
        x = 5

    func_inner()
    print('Локальное x сменилось на', x)

func_outer()
```

Когда мы находимся внутри func_inner, переменная x, определённая в первой строке func_outer находится ни в локальной области видимости (определение переменной не входит в блок func_inner), ни в глобальной области видимости (она также и не в основном блоке программы). Мы объявляем, что хотим использовать именно эту переменную x, следующим образом: nonlocal x.

Попробуйте заменить "nonlocal x" на "global x", а затем удалить это зарезервированное слово, и пронаблюдайте за разницей между этими двумя случаями.

Значения аргументов по умолчанию

Зачастую часть параметров функций могут быть необязательными, и для них будут использоваться некоторые заданные значения по умолчанию, если пользователь не укажет собственных. Этого можно достичь с помощью значений аргументов по умолчанию. Их можно указать, добавив к имени параметра в определении функции оператор присваивания (=) с последующим значением.

Обратите внимание, что значение по умолчанию должно быть константой. Или точнее говоря, оно должно быть неизменным.

```
def say(message, times = 1):
    print(message * times)

say('Привет')
say('Мир', 5)
```

Функция под именем say используется для вывода на экран строки указанное число раз. Если мы не указываем значения, по умолчанию строка выводится один раз. Мы достигаем этого указанием значения аргумента по умолчанию, равного 1 для параметра times.

При первом вызове say мы указываем только строку, и функция выводит её один раз. При втором вызове say мы указываем также и аргумент 5, обозначая таким образом, что мы хотим сказать фразу 5 раз.

Важно! Значениями по умолчанию могут быть снабжены только параметры, находящиеся в конце списка параметров. Таким образом, в списке параметров функции параметр со значением по умолчанию не может предшествовать параметру без значения по умолчанию. Это связано с тем, что значения присваиваются параметрам в соответствии с их положением. Например, def func(a, b=5) допустимо, а def func(a=5, b) – недопустимо.

Ключевые аргументы

Если имеется некоторая функция с большим числом параметров, и при её вызове требуется указать только некоторые из них, значения этих параметров могут задаваться по их имени – это называется ключевые параметры. В этом случае для передачи аргументов функции используется имя (ключ) вместо позиции (как было до сих пор).

Есть два преимущества такого подхода: во-первых, использование функции становится легче, поскольку нет необходимости отслеживать порядок аргументов; во-вторых, можно задавать значения только некоторым избранным аргументам, при условии, что остальные параметры имеют значения аргумента по умолчанию.

```
def func(a, b=5, c=10):
    print('a равно', a, ', b равно', b, ', a c равно', c)

func(3, 7)
func(25, c=24)
func(c=50, a=100)
```

Функция с именем func имеет один параметр без значения по умолчанию, за которым следуют два параметра со значениями по умолчанию.

При первом вызове, func(3, 7), параметр a получает значение 3, параметр b получает значение 7, а c получает своё значение по умолчанию, равное 10.

При втором вызове func(25, c=24) переменная a получает значение 25 в силу позиции аргумента. После этого параметр c получает значение 24 по имени, т.е. как ключевой параметр. Переменная b получает значение по умолчанию, равное 5.

При третьем обращении func(c=50, a=100) мы используем ключевые аргументы для всех указанных значений. Обратите внимание на то, что мы указываем значение для параметра c перед значением для a, даже несмотря на то, что в определении функции параметр a указан раньше c.

Переменное число параметров

Иногда бывает нужно определить функцию, способную принимать любое число параметров. Этого можно достичь при помощи звёздочек:

```
def total(a=5, *numbers, **phonebook):
    print('a', a)
    #проход по всем элементам кортежа
    for single_item in numbers:
    print('single_item', single_item)
    #проход по всем элементам словаря
    for first_part, second_part in phonebook.items():
print(first_part,second_part)

print(total(10,1,2,3,Jack=1123,John=2231,Inge=1560))
```

Когда мы объявляем параметр со звёздочкой (например, *param), все позиционные аргументы начиная с этой позиции и до конца будут собраны в кортеж под именем param.

Аналогично, когда мы объявляем параметры с двумя звёздочками (**param), все ключевые аргументы начиная с этой позиции и до конца будут собраны в словарь под именем param.

Только ключевые параметры

Если некоторые ключевые параметры должны быть доступны только по ключу, а не как позиционные аргументы, их можно объявить после параметра со звёздочкой:

```
def total(initial=5, *numbers, extra_number):
    count = initial
    for number in numbers:
    count += number
    count += extra_number
    print(count)

total(10, 1, 2, 3, extra_number=50)
total(10, 1, 2, 3)
# Вызовет ошибку, поскольку мы не указали значение
# аргумента по умолчанию для 'extra_number'.
```

Объявление параметров после параметра со звёздочкой даёт только ключевые аргументы. Если для таких аргументов не указано значение по умолчанию, и оно не передано при вызове, обращение к функции вызовет ошибку, в чём мы только что убедились.

Обратите внимание на использование +=, который представляет собой сокращённый оператор, позволяющий вместо x = x + y просто написать x += y.

Если вам нужны аргументы, передаваемые только по ключу, но не нужен параметр со звёздочкой, то можно просто указать одну звёздочку без указания имени: def total(initial=5, *, extra_number).

Оператор "return"

Оператор return используется для возврата из функции, т.е. для прекращения её работы и выхода из неё. При этом можно также вернуть некоторое значение из функции.

```
def maximum(x, y):
    if x > y:
    return x
    elif x == y:
    return 'Числа равны.'
    else:
    return y

print(maximum(2, 3))
```

Функция maximum возвращает максимальный из двух параметров, которые в данном случае передаются ей при вызове. Она использует обычный условный оператор if..else для определения наибольшего числа, а затем возвращает это число.

Обратите внимание, что оператор return без указания возвращаемого значения эквивалентен выражению return None. None – это специальный тип данных в Python, обозначающий ничего. К примеру, если значение переменной установлено в None, это означает, что ей не присвоено никакого значения.

Каждая функция содержит в неявной форме оператор return None в конце, если вы не указали своего собственного оператора return. В этом можно убедиться, запустив print(someFunction()), где функция someFunction – это какая-нибудь функция, не имеющая оператора return в явном виде. Например:

```
def someFunction():
    pass
```

Оператор pass используется в Python для обозначения пустого блока команд.

Строки документации

Python имеет остроумную особенность, называемую строками документации, обычно обозначаемую сокращённо docstrings. Это очень важный инструмент, которым вы обязательно должны пользоваться, поскольку он помогает лучше документировать программу и облегчает её понимание. Поразительно, но строку документации можно получить, например, из функции, даже во время выполнения программы!

```
def printMax(x, y):
    '''Выводит максимальное из двух чисел.

       Оба значения должны быть целыми числами.'''

    x = int(x) # конвертируем в целые, если возможно
   y = int(y)
    if x > y:
        print(x, 'наибольшее')
    else:
        print(y, 'наибольшее')

printMax(3, 5)
print(printMax.__doc__)
```

Строка в первой логической строке функции является строкой документации для этой функции. Обратите внимание на то, что строки документации применимы также к модулям и классам, о которых мы узнаем в соответствующих главах.

Строки документации принято записывать в форме многострочной строки, где первая строка начинается с заглавной буквы и заканчивается точкой. Вторая строка оставляется пустой, а подробное описание начинается с третьей. Вам настоятельно рекомендуется следовать такому формату для всех строк документации всех ваших нетривиальных функций.

Доступ к строке документации функции printMax можно получить с помощью атрибута этой функции (т.е. имени, принадлежащего ей) __doc__ (обратите внимание на двойное

подчёркивание). Просто помните, что Python представляет всё в виде объектов, включая функции.

Если вы пользовались функцией help() в Python, значит вы уже видели строки документации. Эта функция просто-напросто считывает атрибут __doc__ соответствующей функции и аккуратно выводит его на экран. Вы можете проверить её на рассмотренной выше функции: просто включите help(printMax) в текст программы.

Точно так же автоматические инструменты могут получать документацию из программы. Именно поэтому настоятельно рекомендуется использовать строки документации для любой нетривиальной функции, которую вы пишете.

Функция-генератор

Функция, возвращающая генератор.

Выглядят функции-генераторы также как и обычные, но содержат выражения с ключевым словом yield для последовательного генерирования значений, которые могут быть использованы в циклах "for in", либо их получения при помощи функции next.

```
def my_animal_generator():
    yield 'корова'
    for animal in ['кот', 'собака', 'медведь']:
yield animal
    yield 'кит'


for animal in my_animal_generator():
    print(animal)
```

На каждой yield работа функции временно приостанавливается, при этом сохраняется состояние исполнения, включая локальные переменные, указатель на текущую инструкцию, внутренний стек и состояние обработки исключения. При последующем обращении к итератору генератора (при вызовах его методов) функция продолжает своё

исполнение с места, на котором была приостановлена. Этим функции-генераторы отличаются от обычных функций, при вызове которых исполнение всякий раз начинается с начала.

Если функция достигает инструкции return, либо конца (без указания упомянутой инструкции), возбуждается исключение StopIteration и итератор исчерпывает себя.

Модуль 4

Модули и пакеты

Модули

Как можно использовать код повторно, помещая его в функции, мы уже видели. А что, если нам понадобится повторно использовать различные функции в других наших программах? Как вы уже, наверное, догадались, ответ – модули.

Существуют разные способы составления модулей, но самый простой – это создать файл с расширением .py, содержащий функции и переменные.

Другой способ – написать модуль на том языке программирования, на котором написан сам интерпретатор Python. Например, можно писать модули на языке программирования C, которые после компиляции могут использоваться стандартным интерпретатором Python.

Модуль можно импортировать в другую программу, чтобы использовать функции из него. Точно так же мы используем стандартную библиотеку Python. Сперва посмотрим, как использовать модули стандартной библиотеки.

```
import random

s = "abcdef"

letter = random.choice(s)

print(letter)
```

В начале мы импортируем модуль random командой import. Этим мы говорим Python, что хотим использовать этот модуль.

Когда Python выполняет команду import random, он ищет модуль random. В данном случае это один из встроенных модулей, и Python знает, где его искать.

Если бы это был не скомпилированный модуль, т.е. модуль, написанный на Python, тогда интерпретатор Python искал бы его в каталогах, перечисленных в переменной sys.path. Если модуль найден, выполняются команды в теле модуля, и он становится доступным.

Обратите внимание, что инициализация происходит только при первом импорте модуля.

Доступ к функции choice в модуле random предоставляется при помощи точки, т.е. random.choice(). Это явно показывает, что это имя является частью модуля random. Ещё одним преимуществом такого обозначения является то, что имя не конфликтует с именем функции choice, которая может использоваться в вашей программе.

Файлы байткода .pyc

Импорт модуля – относительно дорогостоящее мероприятие, поэтому Python предпринимает некоторые трюки для ускорения этого процесса. Один из способов – создать байт-компилированные файлы (или байткод) с расширением .pyc, которые являются некой промежуточной формой, в которую Python переводит программу. Такой файл .pyc полезен при импорте модуля в следующий раз в другую программу – это произойдёт намного быстрее, поскольку значительная часть обработки, требуемой при импорте модуля, будет уже проделана. Этот байткод также является платформо-независимым.

Оператор from ... import ...

Чтобы импортировать функцию choice прямо в программу и не писать всякий раз random. при обращении к ней, можно воспользоваться выражением "from random import choice".

Для импорта всех имён, использующихся в модуле random, можно выполнить команду "from random import *". Это работает для любых модулей.

В общем случае вам следует избегать использования этого оператора и использовать вместо этого оператор import, чтобы предотвратить конфликты имён и не затруднять чтение программы.

Имя модуля – __name__

У каждого модуля есть имя, и команды в модуле могут узнать имя их модуля. Это полезно, когда нужно знать, запущен ли модуль как самостоятельная программа или импортирован. Как уже упоминалось выше, когда модуль импортируется впервые, содержащийся в нём код исполняется. Мы можем воспользоваться этим для того, чтобы заставить модуль вести себя по-разному в зависимости от того, используется ли он сам по себе или импортируется в другую программа. Этого можно достичь с применением атрибута модуля под названием __name__.

```
if __name__ == '__main__':
    print('Эта программа запущена сама по себе.')
else:
    print('Меня импортировали в другой модуль.')
```

В каждом модуле Python определено его имя – __name__. Если оно равно '__main__', это означает, что модуль запущен самостоятельно пользователем, и мы можем выполнить соответствующие действия.

Создание собственных модулей

Создать собственный модуль очень легко. Да вы всё время делали это! Ведь каждая программа на Python также является и модулем. Необходимо лишь убедиться, что у неё установлено расширение .py. Следующий пример объяснит это:

```
def sayhi():
    print('Привет! Это говорит мой модуль.')

__version__ = '0.1'

# Конец модуля mymodule.py
```

Выше приведён простой модуль. Как видно, в нём нет ничего особенного по сравнению с обычной программой на Python. Далее посмотрим, как использовать этот модуль в других наших программах.

Помните, что модуль должен находиться либо в том же каталоге, что и программа, в которую мы импортируем его, либо в одном из каталогов, указанных в sys.path.

Ещё один модуль:

```
import mymodule

mymodule.sayhi()

print ('Версия', mymodule.__version__)
```

Обратите внимание, что мы используем всё то же обозначение точкой для доступа к элементам модуля. Python повсеместно использует одно и то же обозначение точкой, придавая ему таким образом характерный "Python-овый" вид и не вынуждая нас изучать всё новые и новые способы делать что-либо.

Вот версия, использующая синтаксис from..import:

```
from mymodule import sayhi, __version__

sayhi()

print('Версия', __version__)
```

Обратите внимание, что если в модуле, импортирующем данный модуль, уже было объявлено имя __version__, возникнет конфликт. Это весьма вероятно, так как объявлять версию любого модуля при помощи этого имени – общепринятая практика. Поэтому всегда рекомендуется отдавать предпочтение оператору import, хотя это и сделает вашу программу немного длиннее.

Вы могли бы также использовать:

```
from mymodule import *
```

Это импортирует все публичные имена, такие как sayhi, но не импортирует __version__, потому что оно начинается с двойного подчёркивания

Дзэн Python Одним из руководящих принципов в Python является "Явное лучше Неявного". Выполните команду “import this”, чтобы узнать больше о принципах Python.

Функция dir

Встроенная функция dir() возвращает список имён, определяемых объектом. Например, для модуля в этот список входят функции, классы и переменные, определённые в этом модуле.

Эта функция может принимать аргументы. Если в качестве аргумента указано имя модуля, она возвращает список имён, определённых в этом модуле. Если никакого аргумента не передавать, она вернёт список имён, определённых в текущем модуле.

```
>>> import sys # получим список атрибутов модуля 'sys'
>>> dir(sys)
['__displayhook__', '__doc__',... ]
>>> dir() # получим список атрибутов текущего модуля
['__builtins__', '__doc__', '__name__', '__package__', 'sys'] >>> a
= 5 # создадим новую переменную 'a'
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'a', 'sys']
>>> del a # удалим имя 'a'
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'sys']
```

Сперва мы видим результат применения dir к импортированному модулю sys. Видим огромный список атрибутов, содержащихся в нём.

Затем мы вызываем функцию dir, не передавая ей параметров. По умолчанию, она возвращает список атрибутов текущего модуля. Обратите внимание, что список импортированных модулей также входит туда.

Чтобы пронаблюдать за действием dir, мы определяем новую переменную a и присваиваем ей значение, а затем снова вызываем dir. Видим, что в полученном списке появилось дополнительное значение. Удалим переменную/атрибут из текущего модуля при помощи оператора del, и изменения вновь отобразятся на выводе функции dir.

Замечание по поводу del: этот оператор используется для удаления переменной/имени, и после его выполнения, в данном случае – del a, к переменной a больше невозможно обратиться – её как будто никогда и не было.

Обратите внимание, что функция dir() работает для любого объекта. Например, выполните "dir('print')", чтобы увидеть атрибуты функции print, или "dir(str)", чтобы увидеть атрибуты класса str.

Пакеты

К настоящему времени вы, вероятно, начали наблюдать некоторую иерархию в организации ваших программ. Переменные обычно находятся в функциях. Функции и глобальные переменные обычно находятся в модулях. А что, если возникнет необходимость как-то организовать модули? Вот здесь-то и выходят на сцену пакеты.

Пакеты – это просто каталоги с модулями и специальным файлом __init__.py, который показывает Python, что этот каталог особый, так как содержит модули Python.

Представим, что мы хотим создать пакет под названием "world" с подпакетами "asia", "africa" и т.д., которые, в свою очередь, будут содержать модули "india", "madagascar" и т.д.

Для этого следовало бы создать следующую структуру каталогов:

```
| - <некоторый каталог из sys.path>/
| |---- world/
|      |---- __init__.py
|      |---- asia/
|      |    |---- __init__.py
|      |    |---- india/
|      |         |---- __init__.py
|      |         |---- foo.py
|      |---- africa/
|           |---- __init__.py
|           |---- madagascar/
|                |---- __init__.py
|                |---- bar.py
```

Пакеты – это удобный способ иерархически организовать модули. Такое часто встречается в стандартной библиотеке.

Установка библиотек

В Каталоге пакетов Python (https://pypi.python.org/pypi) существует колоссальное количество открытых библиотек, которые вы можете использовать в своих программах. Для их установки можно воспользоваться pip (https://pip.pypa.io//en/latest/).

Модуль 5

Практическая работа 1

Модуль 6

Работа с файловой системой

Работа с файлами

До сих пор наши программы ничего не сохраняли в результате своей работы. Как правило они работали недолго, входные данные либо создавались самой программой, либо запрашивались у пользователя. Результат работы выводился в консоль и следующий запуск программы начинался с чистого листа.

Другие программы, которые работают продолжительное время, могут сохранять результат работы таким образом, чтобы его можно было его использовать после перезапуска программы.

Режимы работы с файлами.

Для работы с файлом нужно его открыть. Эту операцию выполняет функция open(filename, mode).

Первый параметр (строка) определяет имя открываемого для работы файла и может содержать или только имя файла (тогда он ищется в текущем каталоге) или его абсолютное имя (вместе с путём, где этот файл находится). Параметр mode определяет способ работы с открываемым файлом:

- чтение (mode='r'): файл только читается, изменять его нельзя
- чтение/запись (mode='r+'): файл можно читать и изменять
- запись (mode='w'): файл можно только изменять. При этом его содержимое удаляется, то есть файл перезаписывается.
- запись/чтение (mode='w+'): Что и режим 'w', но данные можно также читать.
- запись (mode='a'): Файл открывается на запись, при этом курсор находится в последней позиции
- запись/чтение (mode='a+'): Что и режим 'a', но данные можно также читать.

Для открытия файла в бинарном режиме необходимо добавить 'b', например 'rb'

Примеры использования:

```
open('text.txt', 'r')
# файл text.txt из текущего каталога открывается на чтение
open('c:\python\data\test.log', 'w')
# файл test.log из каталога c:\python\data\ открывается на запись
```

Для работы с файлами в кодировке юникод, особенно, если в нём содержаться символы национальных алфавитов, необходимо указать кодировку:

```
open('text.txt', 'r', encoding='utf-8')
open('text.txt', encoding='utf-8', mode='w')
```

Чтение из файла

Как было отмечено ранее, работа с файлом начинается с его открытия в одном из режимов.

Функция open возвращает объект, обращаясь к которому мы и будем выполнять все необходимые манипуляции с файлом. После работы с файлом его необходимо закрыть. Сделать это можно с помощью метода close(). Однако в языке Python можно оформить открытие файла в виде отдельного блока с помощью оператора with, по окончании выполнения которого файл автоматически закроется:

```
with open('test.txt') as f:
    # что-то делаем
    # что-то делаем
```

Строчка, начинающаяся со служебного слова with является заголовком блока, все операторы которого должны быть записаны ниже со стандартным отступом. По окончании блока файл автоматически закроется.

Для чтения файла в Python используются следующие методы (каждый вызывается для объекта файла):

f.read(n)

Параметр n целое число. Если он указан, то метод возвращает строку, состоящую из следующих n символов файла. Если параметр не указан, то в строку считывается весь файл целиком, начиная с текущей позиции, т.е. с того символа, до которого файл был прочитан к настоящему моменту.

```
with open('test.txt') as f:
    print(f.read(10)) # Читаем первые 10 символов
   print(f.read(5)) # Читаем следующие 5 символов
   print(f.read()) # Читаем все оставшиеся символы
```

f.readline()

Метод читает одну строку (до символа новой строки '\n').

Возвращается прочитанная строка, включая символ новой строки. Например, ниже показано, как вывести на печать файл с нумерацией его строк:

```
i = 1
with open('test.txt') as f:
    s = f.readline()
    while s:
    print('{0:d} {1:s}'.format(i, s), end = '')   i
+= 1
        s = f.readline()
```

f.readlines()

Метод читает весь файл построчно.

Возвращает список строк (включая символы новой строки).

Помимо указанных методов работы с файлами в языке Python существует следующая удобная конструкция, позволяющая перебирать строки файла. Тогда пример с печатью нумерованных строк файла можно записать короче и естественней:

```
i = 1
with open('test.txt') as f:
    for line in f:
    print('{0:d} {1:s}'.format(i, s), end = '')   i
+= 1
```

Обратите внимание, что нам не надо зачитывать строки методами. Файловый объект в цикле for сам возвращает строку на каждой итерации

Запись в файл

Для записи в файл надо открыть файл для записи в соответствующих режимах.

Рассмотрим пример:

```
# открываем файл на запись и записываем строку
with open('output.txt', 'w') as f:
    f.write('Line one')
# открываем файл на дозапись и записываем строку в конец файла
with open('output.txt', 'a') as f:
    f.write('\nLine two')
```

Другие методы для работы с файлом

f.tell()

возвращает целое, представляющее собой текущую позицию в файле f, измеренную в байтах от начала файла.

f.seek(смещение, откуда).

Изменяет позицию курсора. Позиция вычисляется прибавлением смещения к точке отсчёта. Точка отсчёта выбирается из параметра откуда. Значение 0 параметра откуда отмеряет смещение от начала файла, значение 1 применяет текущую позицию в файле, а значение 2 в качестве точки отсчёта использует конец файла. Параметр откуда может быть опущен и по умолчанию устанавливается в 0, используя начало файла в качестве точки отсчёта.

В текстовых файлах работает только смещение от начала файла, то есть значение 0. Для того, чтобы использовать другие параметры “откуда” необходимо открыть файл в бинарном режиме добавив символ “b”, например: “rb” - открывает файл на чтение в бинарном режиме.

Атрибутыобъектовфайлов

- f.closed
  Логическое значение, соответствующее состоянию файла: False – файл открыт, True – закрыт.
- f.mode
  Режим ввода-вывода.
- f.name
  Имя файла.
- f.encoding
  Строка с названием кодировки файла, если определена (например, 'latin-1' или 'utf-8').

Работа с данными в формате CSV

Для работы со специфическими форматами данных Python предоставляет соответствующие модули. Например, для того чтобы работать с данными в формате CSV, необходимо подключить модуль **csv**.

Пример чтения файла users.csv, содержимое которого выглядит как:
Mike,Dow,33,456-34-12

```
import csv
# открываем файл на чтение
with open('users.csv', 'r', encoding='utf-8' ) as f:
    # создаём объект, указывая в качестве разделителя запятую
    reader = csv.reader(f, delimiter=',')
    # читаем файл построчно
    for row in reader:
    # в row приходит строка в виде списка строковых данных   #
['Mike', 'Dow', '33', '456-34-12']
```

Запись в файл происходит аналогичным образом:

```
import csv
# открываем файл на дозапись
with open('users.csv', 'a', newline='' ) as f:
    # создаём объект, указывая в качестве разделителя запятую
    writer = csv.writer(f, delimiter=',')
    # Дописываем строку в файл
    writer.writerow(['John', 'Smith', 25, '123-45-67'])
```

Работа с файловой системой

Стандартный модуль os имеет интерфейс работы с файловой системой. Каждая программа имеет текущий каталог. Функция os.getcwd возвращает текущий каталог:

```
import os
cwd = os.getcwd()
print(cwd)
```

Проверить наличие файла в текущем каталоге:

```
os.path.exists('my_file')
```

Вывести список файлов и подкаталогов для данного каталога:

```
os.listdir(path)
```

Следующий пример выводит список всех файлов и подкаталогов для данного каталога:

```
import os
for name in os.listdir(dir):
    path = os.path.join(dir, name)
    if os.path.isfile(path):
    print(path)
    else:
        print('[' + path + ']')
```

Модуль os

предоставляет множество функций для работы с операционной системой. Вот наиболее часто используемые из них.

- os.name - имя операционной системы.
- os.environ - словарь переменных окружения. Изменяемый (можно добавлять и удалять переменные окружения).
- os.chdir(path) - смена текущей директории.
- os.getcwd() - текущая рабочая директория.
- os.listdir(path=".") - список файлов и директорий в папке.
- os.mkdir(path) - создаёт директорию. Ошибка, если директория существует.
- os.remove(path) - удаляет путь к файлу.
- os.rename(src, dst) - переименовывает файл или директорию из src в dst.
- os.rmdir(path) - удаляет пустую директорию.
- os.truncate(path, length) - обрезает файл до длины length.
- os.path - модуль, реализующий некоторые полезные функции на работы с путями.

Модуль os.path

Модуль является вложенным модулем в модуль os, и реализует некоторые полезные функции на работы с путями.

- os.path.abspath(path) - возвращает нормализованный абсолютный путь.
- os.path.basename(path) - базовое имя пути (эквивалентно os.path.split(path)[1]).
- os.path.dirname(path) - возвращает имя директории пути path.
- os.path.exists(path) - возвращает True, если path указывает на существующий путь или дескриптор открытого файла.
- os.path.getatime(path) - время последнего доступа к файлу, в секундах.
- os.path.getmtime(path) - время последнего изменения файла, в секундах.
- os.path.getctime(path) - время создания файла (Windows), время последнего изменения файла (Unix).
- os.path.getsize(path) - размер файла в байтах.
- os.path.isabs(path) - является ли путь абсолютным.
- os.path.isfile(path) - является ли путь файлом.
- os.path.isdir(path) - является ли путь директорией.

- os.path.join(path1[, path2[, ...]]) - соединяет пути с учётом особенностей операционной системы.
- os.path.normcase(path) - нормализует регистр пути (на файловых системах, не учитывающих регистр, приводит путь к нижнему регистру).
- os.path.normpath(path) - нормализует путь, убирая избыточные разделители и ссылки на предыдущие директории. На Windows преобразует прямые слеши в обратные.
- os.path.realpath(path) - возвращает канонический путь, убирая все символические ссылки (если они поддерживаются).
- os.path.relpath(path, start=None) - вычисляет путь относительно директории start (по умолчанию - относительно текущей директории).
- os.path.samefile(path1, path2) - указывают ли path1 и path2 на один и тот же файл или директорию.
- os.path.sameopenfile(fp1, fp2) - указывают ли дескрипторы fp1 и fp2 на один и тот же открытый файл.
- os.path.split(path) - разбивает путь на кортеж (голова, хвост), где хвост - последний компонент пути, а голова - всё остальное. Хвост никогда не начинается со слеша (если путь заканчивается слешем, то хвост пустой). Если слешей в пути нет, то пустой будет голова.
- os.path.splitext(path) - разбивает путь на пару (root, ext), где ext начинается с точки и содержит не более одной точки.

Разбор параметров командной строки

Консольные программы часто используют параметры запуска, задаваемые через командную строку.

При запуске программы на языке Python параметры командной строки сохраняются в списке sys.argv. Первый элемент этого списка – имя программы. Остальные элементы – это параметры, указанные в командной строке после имени программы.

Например, если запустить в консоли скрипт по имени params.py как:

```
>python params.py hello world
```

то в самом скрипте мы сможем отловить все параметры следующим образом:

```
import sys


script_name = sys.argv[0]
# имя сценария, выполняемого в данный момент (params.py)
param1 = sys.argv[1]
# первый параметр (hello)
param2 = sys.argv[2]
# второй параметр (world)
```

В простых случаях параметры командной строки можно обрабатывать вручную, однако сложную обработку параметров командной строки лучше производить с помощью модуля argparse: (https://docs.python.org/3/howto/argparse.html)

Модуль 7

Исключения и обработка ошибок

Ошибки и исключения

До этого момента сообщения об ошибках лишь упоминались, но если вы пробовали примеры на практике — возможно, вы уже видели некоторые. Существует (как минимум) два различимых вида ошибок: синтаксические ошибки (syntax errors) и исключения (exceptions).

Ошибки

Синтаксические ошибки, также известные как ошибки разбора кода (парсинга, parsing) — вероятно, наиболее привычный вид жалоб компилятора, попадающихся вам при изучении Python:

```
>>> while True print('Hello world')
     File "<stdin>", line 1, in ?
     while True print('Hello world')
     ^
     SyntaxError: invalid syntax
```

Парсер повторно выводит ошибочную строку и отображает небольшую «стрелку», указывающую на самую первую позицию в строке, где была обнаружена ошибка. Причина ошибки (или по крайней мере место обнаружения) находится в символе, предшествующем указанному: в приведённом примере ошибка обнаружена на месте вызова функции print(), поскольку перед ним пропущено двоеточие (':'). Также здесь выводятся имя файла и номер строки, благодаря этому вы знаете в каком месте искать, если ввод был сделан из сценария.

Рассмотрим ещё один пример. Что, если мы ошибочно напишем print как Print? Обратите внимание на заглавную букву. В этом случае Python опять поднимает синтаксическую ошибку:

```
>>> Print('Привет, Мир!')
    Traceback (most recent call last):
    File "<pyshell#0>", line 1, in <module>
    Print('Привет, Мир!')
    NameError: name 'Print' is not defined
```

Обратите внимание, что была поднята ошибка NameError, а также указано место, где была обнаружена ошибка. Так в данном случае действует обработчик ошибок.

Исключения

Даже если выражение или оператор синтаксически верны, они могут вызвать ошибку при попытке их исполнения. Ошибки, обнаруженные при исполнении, называются исключениями (exceptions).

Исключения возникают тогда, когда в программе возникает некоторая исключительная ситуация. Например, к чему приведёт попытка чтения несуществующего файла? Или если файл был случайно удалён, пока программа работала? Это касается и программ, содержащих недействительные команды. В этом случае Python поднимает руки и сообщает, что обнаружил ошибку.

Исключения не фатальны: позже вы научитесь перехватывать их в программах на Python. Большинство исключений, правда, как правило, не обрабатываются программами и приводят к сообщениям об ошибке, таким как следующие:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: int division or modulo by zero

>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined

>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: coercing to Unicode: need string or buffer, int found
```

Попытаемся считать что-либо от пользователя. Нажмите Ctrl-D (или Ctrl+Z в Windows) и посмотрите, что произойдёт.

```
>>> s = input('Введите что-нибудь --> ')
Введите что-нибудь →
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
s = input('Введите что-нибудь --> ')
EOFError: EOF when reading a line
```

Python поднимает ошибку с именем EOFError, что означает, что он обнаружил символ конца файла (который вводится при помощи Ctrl-D) там, где не ожидал.

Иерархия исключений

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- Exception
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- EOFError
      +-- ImportError
      +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- NameError
      +-- OSError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- NotADirectoryError

      +-- ReferenceError
```

```
+-- RuntimeError
+-- SyntaxError
|    +-- IndentationError
+-- SystemError
+-- TypeError
+-- ValueError
+-- Warning
```

Описание некоторых типов исключений

- BaseException
  Базовый класс всех исключений
- KeyboardInterrupt
  Возбуждается нажатием клавишей прерывания
- SystemExit
  Завершение программы
- Exception
  Базовый класс для всех исключений, не связанных с завершением программы
- StandardError
  Базовый класс всех исключений, наследующих класс Exception
- ArithmeticError
  Базовый класс исключений, возбуждаемых арифметическими операциями
- FloatingPointError
  Ошибка операции с плавающей точкой
- ZeroDivisionError
  Деление или деления по модулю на ноль
- AttributeError
  Возбуждается при обращении к несуществующему атрибуту
- EnvironmentError
  Ошибка, обусловленная внешними причинами
- IOError
  Ошибка ввода-вывода при работе с файлами
- OSError
  Ошибка операционной системы
- EOFError
  Возбуждается по достижении конца файла
- ImportError
  Ошибка в инструкции import
- LookupError
  Ошибка обращения по индексу или ключу

- IndexError

  Ошибка обращения по индексу за пределами последовательности

- KeyError

  Ошибка обращения к несуществующему ключу словаря

- NameError

  Не удалось отыскать локальное или глобальное имя

- UnboundLocalError

  Ошибка обращения к локальной переменной, которой еще не было присвоено значение

- ReferenceError

  Ошибка обращения к объекту, который уже был уничтожен

- RuntimeError

  Универсальное исключение

- NotImplementedError

  Обращение к нереализованному методу или функции

- SyntaxError

  Синтаксическая ошибка

- IndentationError

  Ошибка оформления отступов

- SystemError

  Нефатальная системная ошибка в интерпретаторе

- TypeError

  Попытка выполнить операцию над аргументом недопустимого типа

- ValueError

  Недопустимый тип

Обработка исключений

Обрабатывать исключения можно при помощи оператора try..except. При этом все обычные команды помещаются внутрь try-блока, а все обработчики исключений – в except-блок.

```
try:
    text = input('Введите что-нибудь --> ')
except EOFError:
    print('Ну зачем вы сделали мне EOF?')
except KeyboardInterrupt:
    print('Вы отменили операцию.')
else:
    print('Вы ввели {0}'.format(text))
```

Вывод:

```
> python try_except.py
Введите что-нибудь -->     # Нажмите ctrl-d
Ну зачем вы сделали мне EOF?
> python try_except.py
Введите что-нибудь -->     # Нажмите ctrl-c
Вы отменили операцию.
> python try_except.py
Введите что-нибудь --> без ошибок
Вы ввели без ошибок
```

Как это работает:

Здесь мы поместили все команды, которые могут вызвать исключения/ошибки, внутрь блока try, а затем поместили обработчики соответствующих ошибок/исключений в блок except. Выражение except может обрабатывать как одиночную ошибку или исключение, так и список ошибок/исключений в скобках. Если не указано имя ошибки или исключения, обрабатываться будут все ошибки и исключения.

Помните, что для каждого выражения try должно быть хотя бы одно соответствующее выражение except. Иначе какой смысл был бы в блоке try?

Если ошибка или исключение не обработано, будет вызван обработчик Python по умолчанию, который останавливает выполнение программы и выводит на экран сообщение об ошибке. Выше мы уже видели это в действии.

Можно также добавить пункт else к соответствующему блоку try..except. Этот пункт будет выполнен тогда, когда исключений не возникает.

В следующем примере мы увидим, как можно получить объект исключения для дальнейшей работы с ним.

Вызов исключения

Исключение можно поднять при помощи оператора raise, передав ему имя ошибки/исключения, а также объект исключения, который нужно выбросить.

Вызываемая ошибка или исключение должна быть классом, который прямо или непрямо является производным от класса Exception.

```
try:
    raise Exception('Всё плохо!')
except Exception as e:
    print(type(e))     # экземпляр исключения
    print(e.args)      # аргументы хранятся в .args
    print(e)           # можно вывести args явно
```

Вывод:

```
<class 'Exception'>
('Всё плохо!')
('Всё плохо!')
```

В блоке except можно указать переменную, следующую за именем исключения. Переменная связывается с экземпляром исключения, аргументы которого хранятся в атрибуте args. Таким образом, вы также можете создать/взять экземпляр исключения перед его порождением и добавить к нему атрибуты по желанию.

Try .. Except

Представим, что в программе происходит чтение файла и необходимо убедиться, что объект файла был корректно закрыт и что не возникло никакого исключения. Этого можно достичь с применением блока finally.

```
import time
try:
    f = open('poem.txt')
    while True: # наш обычный способ читать файлы
line = f.readline()
    if len(line) == 0:
    break
    print(line, end='')
    time.sleep(2) # Пусть подождёт некоторое время
except KeyboardInterrupt:
    print('Вы отменили чтение файла.')
finally:
    f.close()
    print('Очистка: Закрытие файла')
```

Как это работает:

Здесь мы производим обычные операции чтения из файла, но в данном случае добавляем двухсекундный сон после вывода каждой строки при помощи функции time.sleep, чтобы программа выполнялась медленно (ведь Python очень быстр от природы). Во время выполнения программы нажмите ctrl-c, чтобы прервать/отменить выполнение программы.

Пронаблюдайте, как при этом выдаётся исключение KeyboardInterrupt, и программа выходит. Однако, прежде чем программа выйдет, выполняется пункт finally, и файловый объект будет всегда закрыт.

Оператор with

Типичной схемой является запрос некоторого ресурса в блоке try с последующим освобождением этого ресурса в блоке finally. Для того, чтобы сделать это более 'чисто', существует оператор with:

```
with open("poem.txt") as f:
    for line in f:
    print(line, end='')
```

Как это работает:

Вывод должен быть таким же, как и в предыдущем примере. Разница лишь в том, что здесь мы используем функцию open с оператором with – этим мы оставляем автоматическое закрытие файла под ответственность with open. Так что код, который мы бы написали в блоке finally, будет обработан автоматически. Это избавляет нас от необходимости повторно в явном виде указывать операторы try..finally.

Уточнение типа

Так как Python - язык с динамической типизацией, легко допустить ошибку используя для какой-либо операции данные не того типа:

```
n = input('Введите число: ')
print(n * 10)
```

Для избежания подобного рода ошибок, можно воспользоваться возможностью, которая называется "уточнение типа" (type hinting). Работает это так:

```
s: str # подсказали, что в коде появится переменная s типа str # Обратите внимание, что s не существует, это просто подсказка на будущее.
n: int = 10 # создали переменную n типа int
```

Особенно хорошо это работает при определении аргументов функции:

```
def equal(n1: int, n2: int) -> bool:
     return n1 == n2
```

Здесь мы подсказали, что функция equal принимает два аргумента типа int, а возвращает значение типа bool.

Сам Python на уточнение типа никак не реагирует, поэтому это можно считать просто хорошей подсказкой для того, кто читает код. Однако для реальной проверки правильности кода мы можем использовать модуль mypy, который инсталлируется как:

```
> pip install mypy
```

После этого проверку кода можно использовать следующим образом:

```
> python -m mypy наш-скрипт.py
```

Мы сказали интерпретатору, что нужно запустить модуль mypy и передать ему в качестве аргумента наш скрипт. Mypy проанализирует наш код, отслеживая в том числе и уточнение типов. В случае их несовпадения, mypy сообщит об ошибках.

В случае необходимости использовать уточнение типа для сложных объектов, необходимо импортировать модуль typing:

```
from typing import List, Dict, Tuple


l:List[int] # список из значений типа int
d:Dict[str, str] # словарь, у которого и ключи, и значения имеют тип
str
ll:List[Tuple[str, int]]
# список, который состоит из кортежей, которые состоят из строки и
числа
```

Модуль 8

Регулярные выражения

Регулярное выражение

Регулярное выражение — это последовательность символов, используемая для поиска и замены текста в строке или файле.

Для выполнения операций сопоставления с шаблонами регулярных выражений и замены фрагментов строк используется модуль re. Модуль поддерживает операции как со строками Юникода, так и со строками байтов.

Шаблоны регулярных выражений определяются как строки, состоящие из смеси текста и последовательностей специальных символов. В шаблонах часто используются специальные символы и символ обратного слэша, поэтому они обычно оформляются, как «сырые» строки, такие как r'здесь регулярное выражение'. В оставшейся части этого раздела все шаблоны регулярных выражений будут записываться с использованием синтаксиса «сырых» строк.

Синтаксис шаблонов

Ниже приводится список основных последовательностей специальных символов, которые используются в шаблонах регулярных выражений:

- text
  Соответствует строке text.
- .
  Соответствует любому символу, кроме символа перевода строки.
- ^
  Соответствует позиции начала строки.
- $
  Соответствует позиции конца строки.
- *
  Ноль или более повторений предшествующего выражения; соответствует максимально возможному числу повторений.

- +

  Одно или более повторений предшествующего выражения; соответствует максимально возможному числу повторений.

- ?

  Ноль или одно повторение предшествующего выражения.

- {m}

  Соответствует точно m повторениям предшествующего выражения.

- {m, n}

  Соответствует от m до n повторений предшествующего выражения.

- {m,}

  От m до бесконечности.

- [...]

  Соответствует любому символу, присутствующему в множестве, таком как r'[abcdef]' или r'[a-zA-z]'. Специальные символы, такие как *, утрачивают свое специальное значение внутри множества.

- [^...]

  Соответствует любому символу, не присутствующему в множестве, таком как r'[^0-9]'.

- A|B

  Соответствует либо A, либо B, где A и B являются регулярными выражениями.

- (...)

  Подстрока, соответствующая регулярному выражению в круглых скобках, интерпретируется как группа и сохраняется. Содержимое группы может быть получено с помо-щью метода group() объектов класса MatchObject, которые возвращаются операцией поиска совпадений.

Стандартные экранированные последовательности, такие как '\n' и '\t', точно так же интерпретируются и в регулярных выражениях (например, выражению r'\n+' будет соответствовать один или более символов перевода строки). Кроме того, литералы символов, которые в регулярных выражениях имеют специальное значение, можно указывать, предваряя их символом обратного слэша. Например, выражению r'*' соответствует символ *.

Дополнительно ряд экранированных последовательностей, начинающихся символом обратного слэша, соответствуют специальным символам:

- \число
  Соответствует фрагменту текста, совпавшему с группой с указанным номером. Группы нумеруются от 1 до 99, слева направо.
- \A
  Соответствует только началу строки.
- \b
  Соответствует пустой строке в позиции начала или конца слова. Под словом подразумевается последовательность алфавитно-цифровых символов, завершающаяся пробельным или любым другим не алфавитно-цифровым символом.
- \B
  Соответствует пустой строке не в позиции начала или конца слова.
- \d
  Соответствует любой десятичной цифре. То же, что и выражение r'[0-9]'.
- \D
  Соответствует любому нецифровому символу. То же, что и выражение r'[^0-9]'.
- \s
  Соответствует любому пробельному символу. То же, что и выражение r'[\t\n\r\f\v]'.
- \S
  Соответствует любому непробельному символу. То же, что и выражение r'[^\t\n\r\f\v]'.
- \w
  Соответствует любому алфавитно-цифровому символу.
- \W
  Соответствует любому символу, не относящемуся к множеству \w.
- \Z
  Соответствует только концу строки.
- \\
  Соответствует самому символу обратного слэша.

Специальные символы \d, \D, \s, \S, \w и \W интерпретируются иначе при сопоставлении со строками Юникода. В данном случае они совпадают со всеми символами Юникода, соответствующими описанным свойствам. Например, \d совпадает со всеми символами Юникода, которые классифицируются как цифры, будь то европейские, арабские или индийские цифры, каждые из которых занимают различные диапазоны символов Юникода.

Флаги

- A или ASCII
  Сопоставление выполняется только с 8-битными символами ASCII.
- I или IGNORECASE
  Сопоставление выполняется без учета регистра символов.
- L или LOCALE
  При сопоставлении со специальными символами \w, \W, \b и \B используются региональные настройки.
- M или MULTILINE
  Обеспечивает совпадение символов ^ и $ с началом и концом каждой строки в тексте, помимо начала и конца самого текста. (Обычно символы ^ и $ совпадают только с началом и концом всего текста.)
- S или DOTALL
  Обеспечивает совпадение символа точки (.) со всеми символами, включая символ перевода строки.
- X или VERBOSE
  Игнорирует неэкранированные пробельные символы и комментарии в строке шаблона.

Методы модуля re

re.findall(pattern, string, flags=0)
Возвращает список всех неперекрывающихся совпадений с шаблоном pattern в строке string, включая пустые совпадения. Если шаблон имеет группы, возвращает список фрагментов текста, совпавших с группами. Если в шаблоне присутствует более одной группы, каждый элемент в списке будет представлен кортежем, содержащим текст из каждой группы.

re.match(pattern, string, flags=0)
Проверяет наличие совпадения с шаблоном pattern в строке string. В случае успеха возвращает объект типа MatchObject, в противном случае возвращается None.

re.search(pattern, string, flags=0)
Отыскивает в строке string первое совпадение с шаблоном pattern. В случае успеха возвращает объект типа MatchObject; если совпадений не найдено, возвращается None.

re.split(pattern, string, maxsplit=0, flags=0)
Разбивает строку string по совпадениям с шаблоном pattern. Возвращает список строк, включая текст, совпавший с группами, присутствующими в шаблоне. В аргументе maxsplit передается максимальное количество выполняемых разбиений. По умолчанию выполняются все возможные разбиения.

re.subn(pattern, repl, string, count=0, flag=0)
Замещает текстом repl самые первые неперекрывающиеся совпадения с шаблоном pattern в строке string. В аргументе repl допускается использовать обратные ссылки на группы в шаблоне, такие как '\6'. Аргумент count определяет максимальное количество

подстановок. По умолчанию замещаются все найденные совпадения. Возвращает кортеж, содержащий новую строку и количество выполненных подстановок.

re.compile(pattern, flags=0)

Компилирует регулярное выражение для дальнейшего использования с методами данного регулярного выражения.

Объекты MatchObject

Экземпляры класса MatchObject возвращаются методами re.search() и re.match() и содержат информацию о группах, а также о позициях найденных совпадений. Экземпляр mo класса MatchObject обладает следующими методами и атрибутами:

mo.expand(template)

Возвращает строку, полученную заменой совпавших фрагментов экранированными последовательностями в строке template. Обратные ссылки, такие как "\1" и "\2", и именованные ссылки, такие как "\g<n>" и "\g<name>", замещаются содержимым соответствующих групп. Обратите внимание, что эти последовательности должны быть оформлены как «сырые» строки или с дополнительным символом обратного слэша, например: r'\1' или '\\1'.

mo.group([group1, group2, ...])

Возвращает одну или более подгрупп в совпадении. Аргументы определяют номера групп или их имена. Если имя группы не задано, возвращается совпадение целиком. Если указана только одна группа, возвращается строка, содержащая текст, совпавший с группой. В противном случае возвращается кортеж, содержащий совпадения со всеми указанными группами. Если было запрошено содержимое группы с недопустимым именем или номером, возбуждает исключение IndexError.

mo.groups([default])

Возвращает кортеж, содержащий совпадения со всеми группами в шаблоне. В аргументе default указывается значение, возвращаемое для групп, не участвовавших в совпадении (по умолчанию имеет значение None).

mo.groupdict([default])

Возвращает словарь, содержащий совпадения с именованными группами. В аргументе default указывается значение, возвращаемое для групп, не участвовавших в сопоставлении (по умолчанию имеет значение None).

mo.start([group])

mo.end([group])

Эти два метода возвращают индексы начала и конца совпадения с группой в строке. Если аргумент group опущен, возвращаются позиции всего совпадения. Если группа существует, но не участвовала в сопоставлении, возвращается None.

mo.span([group])

Возвращает кортеж из двух элементов (m.start(group), m.end(group)). Если совпадений с группой group не было обнаружено, возвращается кортеж (None, None). Если аргумент group опущен, возвращаются позиции начала и конца всего совпадения.

mo.pos

Значение аргумента pos, переданное методу re.search() или re.match().

mo.endpos

Значение аргумента endpos, переданное методу re.search() или re.match().

mo.lastindex

Числовой индекс последней совпавшей группы. Если ни одна группа не совпала или в шаблоне нет ни одной группы, возвращается None или в шаблоне нет ни одной группы.

mo.re

Объект регулярного выражения, чьим методом re.match() или re.search() был создан данный экземпляр класса MatchObject.

mo.string

Строка, переданная методу re.match() или re.search().

Пример

```
import re
text = "John will be out of the office from 12/15/2012 - 1/3/2013."


# Шаблон регулярного выражения для поиска дат
datepat = re.compile('(\d+)/(\d+)/(\d+)')


# Найти и вывести все даты
for m in datepat.finditer(text):
    print(m.group())


# Отыскать все даты и вывести их в другом формате
monthnames = [None,'Jan','Feb','Mar','Apr','May','Jun',
    'Jul','Aug','Sep','Oct','Nov','Dec'] for m in
datepat.finditer(text):
    s = f'{monthnames[int(m.group(1)]} {m.group(2)}, {m.group(3)}'
    print(s)


# Заменить все даты их значениями в европейском формате
(день/месяц/год)
def fix_date(m):
    return f"{m.group(2)}/{m.group(1)}/{m.group(3)}"
newtext = datepat.sub(fix_date, text)


# Альтернативный способ замены
newtext = datepat.sub(r'\2/\1/\3', text)
```

Модуль 9

Элементы функционального программирования

Функции как объекты

Функции в языке Python – объекты первого класса. Это означает, что они могут передаваться другим функциям в виде аргументов, сохраняться в структурах данных и возвращаться функциями в виде результата.

Ниже приводится пример функции, которая на входе принимает другую функцию и вызывает ее:

```
def callf(fn):
    return fn()

def helloworld():
     return 'Привет, Мир!'

callf(helloworld)    # Передача функции в виде аргумента
```

Lambda выражение

lambda оператор или lambda функция в Python это способ создать анонимную функцию, то есть функцию без имени. Такие функции можно назвать одноразовыми, они используются только при создании. Как правило, lambda функции используются в комбинации с функциями filter, map, reduce.

Синтаксис lambda выражения в Python следующий:

```
lambda arguments: expression
```

В качестве arguments передается список аргументов, разделенных запятой, после чего над переданными аргументами выполняется expression. Если присвоить lambda-функцию переменной, то получим поведение как в обычной функции:

```
>>> multiply = lambda x,y: x * y
>>> multiply(21, 2)
```

Но, конечно же, все преимущества lambda-выражений мы получаем, используя lambda в связке с другими функциями

Функциональные функции

Функция map()

В Python функция map принимает два аргумента: функцию и аргумент составного типа данных, например, список. map применяет к каждому элементу списка переданную функцию. Например, вы прочитали из файла список чисел, изначально все эти числа имеют строковый тип данных, чтобы работать с ними - нужно превратить их в целое число:

```
old_list = ['1', '2', '3', '4', '5', '6', '7']
new_list = []
for item in old_list:
    new_list.append(int(item))
print (new_list)
# [1, 2, 3, 4, 5, 6, 7]
```

Тот же эффект мы можем получить, применив функцию map:

```
old_list = ['1', '2', '3', '4', '5', '6', '7']
new_list = list(map(int, old_list))
print (new_list)
```

Как видите такой способ занимает меньше строк, более читабелен и выполняется быстрее. map также работает и с функциями созданными пользователем:

```
def miles_to_kilometers(num_miles):
    """ Converts miles to the kilometers """
    return num_miles * 1.6
mile_distances = [1.0, 6.5, 17.4, 2.4, 9]
kilometer_distances = list(map(miles_to_kilometers, mile_distances))
print (kilometer_distances)
# [1.6, 10.4, 27.84, 3.84, 14.4]
```

А теперь то же самое, только используя lambda выражение:

```
mile_distances = [1.0, 6.5, 17.4, 2.4, 9]
kilometer_distances = list(map(lambda x: x * 1.6, mile_distances))
print (kilometer_distances)
```

Функция map может быть также применена для нескольких списков, в таком случае функция-аргумент должна принимать количество аргументов, соответствующее количеству списков:

```
l1 = [1,2,3]
l2 = [4,5,6]
new_list = list(map(lambda x,y: x + y, l1, l2))
print (new_list)
# [5, 7, 9]
```

Если же количество элементов в списках совпадать не будет, то выполнение закончится на минимальном списке:

```
l1 = [1,2,3]
l2 = [4,5]
new_list = list(map(lambda x,y:  + y, l1, l2))
print (new_list)
# [5,7]
```

Функция filter()

Функция filter предлагает элегантный вариант фильтрации элементов последовательности. Принимает в качестве аргументов функцию и последовательность, которую необходимо отфильтровать:

```
nums = [34, 65, 23, 7, 44, 87, 25, 17, 88]
filtered = list(filter(lambda x: x > 42, nums))
print (filtered)
# [65, 44, 87, 88]
```

Обратите внимание, что функция, передаваемая в filter должна возвращать значение True / False, чтобы элементы корректно отфильтровались.

Функция reduce()

Функция reduce принимает 2 аргумента: функцию и последовательность. reduce() последовательно применяет функцию-аргумент к элементам списка, возвращает единичное значение. В Python 3 данная функция находится в модуле functools.

Вычисление суммы всех элементов списка при помощи reduce:

```
from functools import reduce

items = [1,2,3,4,5]
sum_all = reduce(lambda x,y: x + y, items)
print (sum_all)
# 15
```

Вычисление наибольшего элемента в списке при помощи reduce:

```
from functolls import reduce

items = [1, 24, 17, 14, 9, 32, 2]
all_max = reduce(lambda a,b: a if (a > b) else b, items)
print (all_max)
# 32
```

Функция zip()

Функция zip объединяет в кортежи элементы из последовательностей переданных в качестве аргументов.

```
a = [1,2,3]
b = "xyz"
c = (None, True)


res = list(zip(a, b, c))
print (res)
# [(1, 'x', None), (2, 'y', True)]
```

Обратите внимание, что zip прекращает выполнение, как только достигнут конец самого короткого списка.

Замыкания

Когда функция интерпретируется как данные, она неявно несет информацию об окружении, в котором была объявлена функция, что оказывает влияние на связывание свободных переменных в функции. В качестве примера рассмотрим модифицированный предыдущий пример, в который были добавлены переменные, а определение функции callf вынесено в отдельный модуль test.py:

```
# test.py
x = 42
def callf(fn):
    return fn()


import test
x = 37
def helloworld():
     return f'Привет, Мир! x = {x}'
test.callf(helloworld)     # Передача функции в виде аргумента
# 'Привет, Мир! x = 37'
```

Обратите внимание, как функция helloworld() в этом примере использует значение переменной x, которая была определена в том же окружении, что и сама функция helloworld(). Однако хотя переменная x определена в файле test.py и именно там фактически вызывается функция helloworld(), при исполнении функцией helloworld() используется не это значение переменной x.

Когда инструкции, составляющие функцию, упаковываются вместе с окружением, в котором они выполняются, получившийся объект называют замыканием. Такое поведение предыдущего примера объясняется наличием у каждой функции атрибута __globals__, ссылающегося на глобальное пространство имен, в котором функция была определена. Это пространство имен всегда соответствует модулю, в котором была объявлена функция.

Когда функция используется как вложенная, в замыкание включается все ее окружение, необходимое для работы внутренней функции. Например:

```
import test
def bar():
    x = 42
    def helloworld():
    return f'Привет, Мир! x = {x}'
    test.callf(helloworld)
    # вернет 'Привет, Мир! x = 42'
```

Замыкания и вложенные функции особенно удобны, когда требуется написать программный код, реализующий концепцию отложенных вычислений. Рассмотрим еще один пример:

```
from urllib.request import urlopen
def page(url):
    def get():
    return urlopen(url).read()
    return get
```

Функция page() в этом примере не выполняет никаких вычислений. Она просто создает и возвращает функцию get(), которая при вызове будет извлекать содержимое веб-страницы. То есть вычисления, которые производятся в функции get(), в действительности откладываются до момента, когда фактически будет вызвана функция get(). Например:

```
>>> python = page('http://www.python.org')
>>> jython = page('http://www.jython.org')
>>> python
<function get at 0x95d5f0>
>>> jython
<function get at 0x9735f0>
>>> pydata = python() # Извлечет страницу http://www.python.org
>>> jydata = jython() # Извлечет страницу http://www.jython.org
```

Две переменные, python и jython, объявленные в этом примере, в действительности являются двумя различными версиями функции get(). Хотя функция page(), которая создала эти значения, больше не выполняется, тем не менее обе версии функции get() неявно несут в себе значения внешних переменных на момент создания функции get(). То есть при выполнении функция get() вызовет urlopen(url) со значением url, которое было передано функции page(). Взглянув на атрибуты объектов python и jython, можно увидеть, какие значения переменных были включены в замыкания. Например:

```
>>> python.__closure__
(<cell at 0x67f50: str object at 0x69230>,)
>>> python.__closure__[0].cell_contents
'http://www.python.org'
>>> jython.__closure__[0].cell_contents
'http://www.jython.org'
```

Замыкание может быть весьма эффективным способом сохранения информации о состоянии между вызовами функции. Например, рассмотрим следующий пример, в котором реализован простой счетчик:

```
def countdown(n):
    def next():
        nonlocal n
        r = n
        n -= 1
        return r
    return next

next = countdown(10)
while True:
    v = next() # Получить следующее значение
    if not v: break
```

В этом примере для хранения значения внутреннего счетчика n используется замыкание. Вложенная функция next() при каждом вызове уменьшает значение счетчика и возвращает его предыдущее значение.

Тот факт, что замыкания сохраняют в себе окружение вложенных функций, делает их удобным инструментом, когда требуется обернуть существующую функцию с целью расширить ее возможности.

Декораторы

Декоратор – это функция, основное назначение которой состоит в том, чтобы служить оберткой для другой функции. Главная цель такого обертывания – изменить или расширить возможности обёртываемого объекта.

Синтаксически декораторы оформляются добавлением специального символа @ к имени, как показано ниже:

```
def add_underscores(fn):
    def wrapper(param):
    return '_' + fn(param) + '_'
    return wrapper

@add_underscores
def say(msg):
    return msg
```

Предыдущий фрагмент является сокращенной версией следующего фрагмента:

```
def say(msg):
     return msg


say = add_underscores(say)
```

В этом примере объявляется функция say(). Однако сразу же вслед за объявлением объект функции передается функции add_underscores(), а возвращаемый ею объект замещает оригинальный объект say.

При использовании декораторы должны помещаться в отдельной строке,

непосредственно перед объявлением функции или класса. Кроме того, допускается указывать более одного декоратора. Например:

```
@foo
@bar
@spam

def func(x):
    pass
```

В этом случае декораторы применяются в порядке следования. Результат получается тот же самый, что и ниже:

```
def func(x):
    pass


func = foo(bar(spam(func)))
```

Кроме того, декоратор может принимать аргументы. Например:

```
@add_symbols('*')
def say(msg):
    return msg
```

При наличии аргументов декораторы имеют следующую семантику:

```
def add_symbols(symbol): # параметр для декоратора
    def decor(fn):
    def wrapper(msg):
    return symbol + fn(msg) + symbol
    return wrapper
    return decor
```

Модуль functools

Модуль functools содержит функции и декораторы, которые удобно использовать для создания высокоуровневых функций и декораторов в функциональном программировании.

```
partial(function [, *args [, **kwargs]])
```

Создает объект типа partial, напоминающий функцию, который при вызове обращается к функции function и передает ей позиционные аргументы args, именованные аргументы kwargs, а также любые дополнительные позиционные и именованные аргументы, переданные объекту при его вызове. Дополнительные позиционные аргументы добавляются в конец args, а дополнительные именованные аргументы добавляются в словарь kwargs, затирая ранее определенные значения с теми же ключами (если таковые имеются).

Обычно функция partial() используется, когда требуется многократно вызвать одну и ту же функцию, большая часть аргументов которой остаются неизменными. Например:

```
from functools import partial

def print_msg(decor, msg):
    return decor + msg + decor

star_msg = partial(print_msg, '*')
underscore_msg = partial(print_msg, '_')

star_msg('hello') # *hello*
star_msg('world') # *world*

underscore_msg('hello') # _hello_
underscore_msg('world') # _world_
```