Урок 4

Полезные инструменты

Передача по ссылке/значению. Вложенные списки (матрицы). Принцип работы операторов and, or. Тернарный оператор if else. Оператор is. Генераторы списков, словарей. Сырые строки (r"") и регулярные выражения. Обработка исключений.

Передача аргументов по ссылке/значению

Рассмотрим такие примеры:

```
>>> n1 = 2
>>> n2 = n1
>>> n1 = 4
>>> print("n1 = ", n1, "n2 = ", n2)
: n1 = 2 n2 = 4
>>> sp1 = [1, 2, 3]
>>> sp2 = sp1
>>> sp2.append(4)
>>> print("sp1 = ", sp1, "sp2 = ", sp2)
: sp1 =  [1, 2, 3, 4] sp2 =  [1, 2, 3, 4]
```

Почему же список l1 тоже изменился, хотя мы изменяли только l2?

Переменная в python - это всего лишь указатель на объект в памяти. Если несколько переменных указывают на один и тот же изменяемый объект, то, изменив объект по одной из ссылок, мы меняем его для всех остальных.

Это особо важно понимать при передаче изменяемых объектов в функцию и при изменении объекта в цикле for in (который итерирует данный объект).

```
def modify(lst):
    lst.append("new")
    return lst
my_list = [1, 2, 3]
mod_list = modify(my_list)
# Функция вернула измененный список
print('mod_list = ', mod_list)
# Но исходный список тоже изменился, подобное неявное поведение нежелательно для функций
print('my_list = ', my_list)
```

```
# Будьте аккуратны при работе с изменяемыми объектами, лучше работайте с их копиями
mod_list = modify(my_list)
my_list = [1, 2, 3]
# Теперь функция получит копию списка и не изменит исходный
mod_list = modify(my_list[:])
print('mod_list = ', mod_list)
print('my_list = ', my_list) # исходный список будет без изменений
```

Теперь рассмотрим изменение списка в процессе перебора его элементов циклом for..in

```
my_list = [1, -2, -4, 0, 5, -2]
# Удаляем все отрицательные элементы
for el in my_list:
    if el < 0:
        my_list.remove(el)
# Думаю, это не тот результат который вы ожидали
print("1)my_list after remove -->", my_list)
# 1) my_list after remove --> [1, -4, 0, 5]
my_list = [1, -2, -4, 0, 5, -2]
# Итерируем по копии, а удаляем из оригинала
for el in my_list[:]:
    if el < 0:
        my_list.remove(el)
# Как хорошо
print("2)my_list after remove -->", my_list)
# 2)my_list after remove --> [1, 0, 5]
```

Используя полный срез [:], вы получите только поверхностную копию, вложенные изменяемые объекты так и останутся ссылками.

```
# Если нужно сделать полную копию со всеми вложенными изменяемыми объектами,
# используем copy
import copy
sp = [[2, 3], [4, 6, [7, 8]]]
sp_copy = copy.deepcopy(sp)
sp[0].append(10)
print('sp_copy = ', sp_copy)
```

Матрицы

Ма́трица — математический объект, записываемый в виде прямоугольной таблицы элементов кольца или поля (например, целых, действительных или комплексных чисел), которая представляет собой совокупность строк и столбцов, на пересечении которых находятся её элементы. Количество строк и столбцов матрицы задаёт размер матрицы. Хотя исторически рассматривались, например, треугольные матрицы[1], в настоящее время говорят исключительно о матрицах прямоугольной формы, так как они являются наиболее удобными и общими.

Матрицы широко применяются в математике для компактной записи систем линейных алгебраических или дифференциальных уравнений. В этом случае количество строк матрицы соответствует числу уравнений, а количество столбцов — количеству неизвестных. В результате решение систем линейных уравнений сводится к операциям над матрицами. Также матрицы применяются в 3D графике, для различных преобразований 3D изображения на плоскость.

В python легко представить матрицу в виде вложенных списков.

```
matrix = [[1, 2, 3],
          [4, 5, 6],
          [7, 8, 9]]
print("matrix = ", matrix)
```

Рассмотрим основные приёмы работы на примере.

```
# Красиво выводим на экран
print("matrix = ")
for line in matrix:
    print(line)
# Работать с элементами матрицы по индексам в python некорректно
print("*************WHILE***************")
i = 0
while len(matrix) > i:
    line = matrix[i]
    j = 0
    while len(line) > j:
        print("matrix[{}][{}] = {}".format(i, j, matrix[i][j]))
        j += 1
    i += 1
#
# Все можно сделать гораздо проще и элегантнее
# Результат один, а код гораздо проще пишется, читается и компактнее
print("*************FOR IN ***************")
for i, line in enumerate(matrix):
    for j, el in enumerate(line):
        print("matrix[{}][{}] = {}".format(i, j, matrix[i][j]))
#
# # Пример транспонирования (поворота) матрицы
print("rotate_matrix = ", list(map(list, zip(*matrix))))
```

Этот пример выведет результат:

```
matrix[0][0] = 1
matrix[0][1] = 2
matrix[0][2] = 3
matrix[1][0] = 4
matrix[1][1] = 5
matrix[1][2] = 6
matrix[2][0] = 7
matrix[2][1] = 8
matrix[2][2] = 9
```

Принцип работы операторов and or

Мы уже использовали операторы and и or при работе с логическими выражениями. На самом деле операторы and и or выполняют булевы операции, но они не возвращают булевы значения: результатом всегда является значение одного из операндов.

Операторы and/or возвращают тот операнд, значение которого определяет результат выражения.

Принцип работы.

```
>>> 5 or 0
: 5
```

Результат операции будет истинным, если ИЛИ левая ИЛИ правая часть будут истинны. Операция читается слева направо, а правый операнд будет True (bool(5) → True). Следовательно, для нахождения результата значение правого операнда вообще не требуется.

```
>>> 0 or 6
: 6
```

Левый операнд будет преобразован False, для результата выражения требуется проверка правого операнда. Вне зависимости от значения правого операнда будет возвращено его значение, т.к. именно оно является решающим.

Аналогично и с оператором and.

```
>>> 0 and 5
: 0
```

Результат будет истинным, если и правая и левая часть будут True. Левый операнд False, значит результат будет False, вычисление правого не требуется.

```
# Практический пример:
# Получить имя, если имени нет, то "безымянный"
# people = {"name": "Вася"}
people = {}
if people.get("name"):
    name = people["name"]
else:
    name = "Безымянный"
print(name)
# Тоже самое с помощью or:
print(people.get("name") or "Безымянный")
# Несмотря на несуществующую переменную u_var, код всё равно выполняется,
# т.к. оператор or не проверяет операнд справа
print(5 or u_var)
```

Тернарный оператор if else

До сих пор вы использовали if else как инструкции для операций ветвления. Но if else также может выступать в качестве оператора, возвращающего значения.

```
d = {"name": "Вася"}
# d = {}
# Пример из предыдущего объяснения:
# Получить имя, если имени нет, то "безымянный"
if d.get("name"):
    name = d["name"]
else:
    name = "Безымянный"
print(name)
# Или с помощью тернарного if else
print(d.get("name") if d.get("name") else "Безымянный")
```

Синтаксис: value_if_logic_true if logic else value_if_logic_false.

Т.е. возвращается или первое значение, или второе, в зависимости от значения выражения после if.

Все операторы делятся на три группы в зависимости от количества операндов.

- Унарные – один операнд.
- Бинарные – два операнда (их большинство).
- Тернарные – три операнда.

Пример унарных операторов:

- not (логический инвертор) – меняет True на False, и наоборот;
- - (унарный минус) - меняет знак результата на противоположный.

Оператор is

Оператор is - возвращает True, если операнды указывают на один и тот же объект в памяти.

```
>>> a = 10
>>> b = 10
>>> a is b
: True
```

Неизменяемые типы данных python кэширует для экономии памяти.

```
>>> c = [1, 2]
>>> d = c
>>> e = [1, 2]
>>> c is d
: True
>>> c is e
: False
```

```
# Так лучше, чем: if a == None
print(a is None)
```

Генераторы списков и словарей

Генераторы - яркий пример “синтаксического сахара” в python. Т.е. всё, что можно сделать с помощью генераторов, можно сделать и без них, это всего лишь синтаксическая конструкция, позволяющая записать частые операции красиво и кратко. Но всё же надо учесть - генераторы обрабатывают результаты несколько быстрее, чем аналогичные им конструкции, реализованные через циклы for in.

```
import random
# Заполняем список произвольными целыми числами
lst = []
for _ in range(10):
    lst.append(random.randint(-10, 10))
print('lst = ', lst)
# Тоже самое, но с помощью генератора списка
# Компактнее код и выполняется быстрее
lst_g = [random.randint(-10, 10) for _ in range(10)]
print('lst_g = ', lst_g)
```

По сути, генератор – это свернутый цикл for in

Обратили внимание на странное имя переменной _ ? Если переменная должна присутствовать в соответствии с синтаксисом, но её значение нигде не используется, то принято такую переменную называть _ (нижнее подчеркивание), это общепринятое соглашение.

```
# Отбрасываем все отрицательные элементы списка
only_positive = [el for el in lst_g if el >= 0]
print('only_positive = ', only_positive)
```

lst_g - список, заполненный в предыдущем примере.
Элемент добавляется в список, если выражение после if True.

Аналогично можно создавать словари с помощью генераторов.

```
# Создаем словарь с помощью генератора словаря
keys = "abcdefg"
values = range(10)
dict_g = {key: value for key, value in zip(keys, values)}
# Подробнее о zip() в следующем примере
print('dict_g = ', dict_g)
# Более простой пример создания словаря генератором
dict2_g = {el: el+4 for el in [1, 4, 6, 8]}
print('dict2_g =', dict2_g)
```

Получим результаты:

```
dict_g =  {'a': 0, 'd': 3, 'g': 6, 'f': 5, 'e': 4, 'c': 2, 'b': 1}
dict2_g = {8: 12, 1: 5, 4: 8, 6: 10}
set_g = {0, 1, 'g', 'b', 'a', -10}
```

Генераторы – очень удобные и мощные инструменты. Важно понимать, что генератор это выражение, которое в качестве результата возвращает определенную последовательность, следовательно, можно использовать генераторы внутри других выражений, но не злоупотребляйте, иначе получите код, сложный для чтения и редактирования.

Регулярные выражения

Сырые строки

Экранированные последовательности позволяют вставить специальные символы, которые сложно ввести с клавиатуры.

Управляющая последовательность	**Описание**
\n	Новая строка (перевод строки)
\r	Возврат каретки
\t	Горизонтальная табуляция
\'	Одинарная кавычка
\"	Двойная кавычка
\\	Обратная косая черта

```
>>> "Hello \t world"
: Hello    world
>>> "Hello \n world"
: Hello
  world
```

"Сырые" строки подавляют экранирование.

Если перед открывающей кавычкой стоит символ 'r' (в любом регистре), то механизм экранирования отключается.

```
>>> r"Hello \t world"
: Hello \t world
```

Regexp

Регулярное выражение — это последовательность символов, используемая для поиска и замены текста в строке или файле. Их поддерживает множество языков общего назначения: Python, Perl, R. Так что изучение регулярных выражений рано или поздно пригодится.

Регулярные выражения используют два типа символов:

- специальные символы: как следует из названия, у этих символов есть специальные значения, например, * означает «любой символ»;
- литералы (например: a, b, 1, 2 и т. д.).

В Python для работы с регулярными выражениями есть модуль re. Для использования его нужно импортировать.

Чаще всего регулярные выражения используются для:

- поиска в строке;
- разбиения строки на подстроки;
- замены части строки.

re.match(pattern, string)

Проверяет, соответствует ли начало строки "string" регулярному выражению "pattern".

Например, выражению 'test' соответствует строка 'test1', но не соответствует строка '1test'. Возвращает объект MatchObject, если строка найдена, или None, если не найдена.

```
string = 'This is a simple test message for test'
string2 = 'test'
pattern1 = 'test$'
pattern2 = '^test'
pattern3 = '^test$'
print(re.search(pattern1, string) is None)   # Строка заканчивается на 'test'
print(re.match(pattern2, string) is None)    # Строка не начинается на 'test'
print(re.match(pattern3, string) is None)    # Строка не является строкой 'test'
print(re.match(pattern3, string2) is None)  # Строка является строкой 'test'
```

Получим:

```
False
True
True
False
```

re.search()

Работает аналогично re.match, но проверяет не только с начала строки, а сканирует строку на совпадения полностью.

re.findall()

Выполняет поиск всех подстрок в строке, соответствующих регулярному выражению.

Возвращает список найденных подстрок, строки не перекрываются.

```
# Сколько раз слово присутствует в строке
string = 'This is a simple test message for test'
found = re.findall(r'test', string)
print(found)
```

Получим:

```
['test', 'test']
```

Так же есть методы:

re.split()

re.sub()

re.compile()

Что, если у нас нет определенного шаблона, и нам надо вернуть набор символов из строки, отвечающей определенным правилам? Такая задача часто стоит при извлечении информации из

строк. Это можно сделать, написав выражение с использованием специальных символов. Вот наиболее часто используемые из них:

Метасимвол	Описание
.	Один любой символ, кроме новой строки \n.
?	0 или 1 вхождение шаблона слева
+	1 и более вхождений шаблона слева
*	0 и более вхождений шаблона слева
\w	Любая цифра или буква (\W — все, кроме буквы или цифры)
\d	Любая цифра [0-9] (\D — все, кроме цифры)
\s	Любой пробельный символ (\S — любой не пробельный символ)
\b	Граница слова
[..]	Один из символов в скобках ([^..] — любой символ, кроме тех, что в скобках)
\	Экранирование специальных символов (\. означает точку или \+ — знак «плюс»)
^ и $	Начало и конец строки соответственно
{n,m}	От n до m вхождений ({,m} — от 0 до m)
a | b	Соответствует a или b
()	Группирует выражение и возвращает найденный текст
\t, \n, \r	Символ табуляции, новой строки и возврата каретки соответственно

```
# Найти все цифры в тексте
pattern = '[0-9]+k'
string = 'If 300 spartans were so brave, so 500 spartans' \
    ' could destroy more than 10k warriors of Darius, but 15k and even 20k'
print(re.findall(pattern, string))
# Найти все диапазоны
pattern2 = '[0-9]+ *- *[0-9]+'
string2 = 'The temperature can be in range 10- 15C next week ' \
          'though it was lesser last week(4 - 9C). It was even ' \
          '-5 some time ago'
print(re.findall(pattern2, string2))
```

Получим:

```
['10k', '15k', '20k']
['10- 15', '4 - 9']
```

Более сложные практические примеры:

```
# Получим нужные данные из лога, для дальнейшего анализа
log = [
    '64 bytes from localhost.localdomain (127.0.0.1): '
    'icmp_req=1 ttl=64 time=0.033 ms',
    '64 bytes from localhost.localdomain (127.0.0.1): '
    'icmp_req=2 ttl=64 time=0.034 ms',
    '64 bytes from localhost.localdomain (127.0.0.1): '
    'icmp_req=3 ttl=64 time=0.031 ms',
    '64 bytes from localhost.localdomain (127.0.0.1): '
    'icmp_req=4 ttl=64 time=0.031 ms']
pattern = re.compile('(icmp_req=[\d]+).*(time=[\d\.]+ ms)')
result = []
for line in log:
    result.append(pattern.search(line).groups())
print(result)
# Извлечём из html-кода только теги
html = '<p style="margin-left:10px;">text' \
       '<b class="super-bold">bold text</b>.</p>'
pattern = '<[^>]+>'
print(re.findall(pattern, html))
```

Подробно и очень доступно о регулярках здесь

Обработка исключений

Типы ошибок

Существует три типа ошибок, которые могут возникнуть в программах:

- синтаксические ошибки (syntax errors);
- ошибки выполнения (runtime errors);
- семантические ошибки (semantic errors).

Чтобы находить и исправлять их быстрее, имеет смысл научиться их различать.

Синтаксические ошибки (syntax errors)

Любой интерпретатор сможет выполнить программу только в том случае, если программа синтаксически правильна. Соответственно, компилятор тоже не сможет преобразовать программу в машинные инструкции, если программа содержит синтаксические ошибки. Когда транслятор находит ошибку (т.е. доходит до инструкции, которую не может понять), он прерывает свою работу и выводит сообщение об ошибке. Для большинства читающих синтаксические ошибки не представляют особой проблемы. Например, часто встречаются стихотворения без знаков препинания, но мы без труда можем их прочесть, хотя это часто порождает неоднозначность их интерпретации. Но трансляторы (и интерпретатор Питона не исключение) очень придирчивы к синтаксическим ошибкам.

Даже если в вашей программе Питон найдет хотя бы незначительную опечатку, он тут же выведет сообщение о том, где он на неё наткнулся, и завершит работу. Такую программу он не сможет выполнить, и поэтому отвергнет. В первые недели вашей практики разработки программ вы, скорее всего, проведете довольно много времени, разыскивая синтаксические ошибки. По мере накопления опыта вы будет допускать их все реже, а находить – всё быстрее.

Ошибки времени выполнения (runtime errors)

Второй тип ошибок обычно возникает во время выполнения программы (их принято называть исключительными ситуациями или, коротко – исключениями (по-английски exceptions). Такие ошибки имеют другую причину. Если в программе возникает исключение, то это означает, что по ходу выполнения произошло что-то непредвиденное: например, программе было передано некорректное значение, или программа попыталась разделить какое-то значение на ноль, что недопустимо с точки зрения дискретной математики. Если операционная система присылает запрос на немедленное завершение программы, то также возникает исключение. Но в простых программах это достаточно редкое явление, поэтому возможно с ними вы столкнетесь не сразу.

Семантические ошибки (semantic errors)

Третий тип ошибок – семантические ошибки. Первым признаком наличия в вашей программе семантической ошибки является то, что она выполняется успешно, т.е. без исключительных ситуаций, но делает не то, что вы от неё ожидаете.

В таких случаях проблема заключается в том, что семантика написанной программы отличается от того, что вы имели в виду. Поиск таких ошибок – задача нетривиальная, т.к. приходится просматривать результаты работу программы и разбираться, что программа делает на самом деле.

runtime errors можно (и нужно) перехватывать и обрабатывать. Ваши программы не должны падать (аварийно завершаться) при некорректных данных/действиях пользователя.

Обработка ошибок

Исключения (exceptions) – ещё один тип данных в python. Исключения необходимы для того, чтобы сообщать программисту об ошибках.

Самый простейший пример исключения - деление на ноль:

```
>>> 100 / 0
Traceback (most recent call last):
  File "", line 1, in
    100 / 0
ZeroDivisionError: division by zero
```

Полный список всех ошибок и их описание можно посмотреть здесь.

Для обработки исключений используется конструкция try - except.

```
# n = 10
n = 'Hello'
try:
    n = int(n)
    print('n успешно преобразована к типу Int')
except ValueError:  # Тип перехватываемого исключения
    print('значение n невозможно преобразовать к типу int')
```

В блоке try мы выполняем инструкцию, которая может породить исключение, а в блоке except мы перехватываем их. Если в блоке try произойдет исключение, то дальнейшие инструкции будут пропущены, а программа перейдёт к выполнению блока except. Если же исключений не возникнет, то блок except выполнен не будет.

Также возможна инструкция except без аргументов, которая перехватывает вообще всё (и прерывание с клавиатуры, и системный выход и т. д.). Поэтому в такой форме инструкция except практически не используется, а используется except Exception. Однако чаще всего перехватывают

исключения по одному, для упрощения отладки (вдруг вы ещё другую ошибку сделаете, а except её перехватит).

Ещё две инструкции, относящиеся к нашей проблеме, это finally и else. Finally выполняет блок инструкций в любом случае, было ли исключение, или нет (применимо, когда нужно непременно что-то сделать, к примеру, закрыть файл). Инструкция else выполняется, если исключения не было.

```
f = open('1.txt')
ints = []
try:
   for line in f:
      ints.append(int(line))
except ValueError:
      print('Это не число. Выходим.')
except Exception:
      print('Это что ещё такое?')
else:
       print('Всё хорошо.')
finally:
      f.close()
      print('Я закрыл файл.')
# Именно в таком порядке: try, группа except, затем else, и только потом finally.
```

Помните инструкцию with as, которую мы использовали для работы с файлами? Она, по сути, является “синтаксическим сахаром” для обработки ошибок, при работе с файлами.

Можно возбуждать (генерировать) и создавать свои собственные исключения, это будет обсуждаться во второй части вебинара.

Домашнее задание

1. Смотреть здесь https://github.com/GeekBrainsTutorial/Python_lessons_basic/tree/master/lesson04

Большинство заданий делятся на три категории easy, normal и hard.

- easy - простенькие задачи, на понимание основ.
- normal - если вы делаете эти задачи, то вы хорошо усвоили урок.
- hard - наиболее хитрые задачи, часто с подвохами, для продвинутых слушателей.

Если вы не можете сделать normal задачи - это повод пересмотреть урок, перечитать методичку и обратиться к преподавателю за помощью.

Если не можете сделать hard – не переживайте, со временем научитесь.

Решение большинства задач будем разбирать в начале каждого вебинара.

Дополнительные материалы

Все то, о чем сказано здесь, но подробнее:

1. Генераторы словарей
2. Сырые строки
3. Регулярные выражения
4. Обработка исключений

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Учим python качественно(habr)
2. Самоучитель по python
3. Книга Лутц М. “Изучаем Python” (4-е издание).
4. Регулярки