Язык программирования

Лекция № 3

Владимир Владимирович Руцкий
rutsky.vladimir@gmail.com

План занятия

- Строки, ввод/вывод, механизм исключений
- Практика

Строки. Повторение (1/2)

- Строки — последовательности:

```
>>> a = "Test!"
>>> a[1]
'e'
>>> [ch for ch in a]
['T', 'e', 's', 't', '!']
>>> b = "0123456789"
>>> b[2:4]
'23'
>>> b[::-1]
'9876543210'
>>> b[::2]
'02468'
```

- Строки — неизменяемые:

```
>>> "test"[1] = "a"
Traceback (most recent call last):
  File "/usr/lib/python3.3/doctest.py", line 1287, in __run
    compileflags, 1), test.globs)
  File "<doctest str_immutable.pycon[1]>", line 1, in <module>
    s[3] = 'a'
TypeError: 'str' object does not support item assignment
```

Строки. Повторение (2/2)

- Различные способы задания:

```
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
>>> "first word \
... second word"
'first word second word'
>>> print("first line\n secondline")
first line
 secondline
>>> r"line with \n in middle"
'line with \\n in middle'
>>> """Multiline with ' or "
... Yes.
... """
'Multiline with \' or "\nYes.\n'
>>>
```

Символы и кодировки

- На экране монитора, бумаге, табличках изображаются **символы**
  - символ: «маленькая кириллическая буква А»
  - изображение: «а»,
- Для хранения символа в памяти компьютера, ему необходимо сопоставить численное представление — **закодировать символ**
- Для кодирования используются различные **кодировки**
- Ранее были распространены однобайтовые кодировки: один символ — один байт
  - Для кириллицы часто использовались `cp1251`, `koi8-r`, `MacCyrillic`
  - Символ: «маленькая кириллическая буква А»: `cp1251` — 0xE0, `koi8-r` — 0xC1, `MacCyrillic` — 0xE0.
- Однобайтные кодировки — ужасны
  - не могут хранить все символы, необходимо переключаться между кодировками для разных языков
  - для файла нужно знать его кодировку, иначе — «кракозябры» (кодировок очень много)

Unicode

- Unicode — стандарт описывающий все символы всех языков (+ иконки, модификаторы и т.п.)
- Каждому символу ставится в соответствие **кодовая позиция** Unicode
  - «маленькая кириллическая буква А» — `U+0430`
- Для хранения кодовых позиций Unicode в памяти компьютера позиции **кодируются**
- Есть ряд **кодировок** хранения кодовых позиций Unicode в памяти (Unicode Transformation Format):
  - UTF-8 — каждый символ кодируется последовательностью от 1 до 6 байт
  - UTF-16 — каждый символ кодируется последовательностью 2 или 4 байта
  - UTF-32 — каждый символ кодируется 4 байтами, числом — кодовой позицией
- код символа «маленькая кириллическая буква А» (`U+0430`)
  - в UTF-8: 0xd0 0xb0,
  - в UTF-16 0x30 0x04,
  - в UTF-32 0x30 0x04 0x00 0x00

Unicode **в** Python

- Строки в Python — это Unicode строки
  - конкретный формат представления в памяти зависит от платформы и настроек компиляции Python
- По умолчанию исходный код файлов считается в UTF-8
  - Это можно изменить, добавив первой строкой комментарий с кодировкой, например:

```
# coding: cp1251
```

```
>>> # В строках можно задавать кодовые позиции Unicode
... print("\u0430 \U00000431 \N{GREEK CAPITAL LETTER DELTA}")
а б Δ
>>> # Получить позицию Unicode:
... ord("г")
1075
>>> # Сконструировать Unicode символ по позиции:
... chr(1075)
'г'
>>> # В строках можно задавать однобайтовые ASCII-коды
... "\x30\x31\x32"
'012'
>>>
```

Байтовые последовательности

```
>>> # Для хранения последовательностей байт используется тип bytes
... b = b'some bytes \xff'
>>> type(b)
<class 'bytes'>
>>> # Задаются также, как строки, но с префиксом 'b'.  Можно задать
... # произвольные байты через '\x..', ASCII символы можно задавать как есть
... b'''\x00\x01 ABC
... def'''
b'\x00\x01 ABC\ndef'
>>> # Элементами bytes являются значения байтов (0--255)
... # Индексация такая же, как в других последовательностях (строках, списках)
... b[0]
115
>>> b[-1]
255
>>> # Строки можно кодировать в набор байт:
... "abc абв".encode('utf-8')
b'abc \xd0\xb0\xd0\xb1\xd0\xb2'
>>> "abc абв".encode('cp1251')
b'abc \xe0\xe1\xe2'
>>> # и обратно
... b'abc \xe0\xe1\xe2'.decode('cp1251')
'abc абв'
>>> b'abc \xe0\xe1\xe2'.decode('utf-8')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe0 in position 4: invalid continuatio
>>> # bytes неизменяемый
```

Работа со строками (1/2)

```
>>> s = "теСТОвая СТРОка"
>>> s = s.capitalize(); s  # Сделать первую букву заглавной, остальные маленькими
'Тестовая строка'
>>> s.count('ст')  # число неперекрывающихся вхождений подстроки
2
>>> s.endswith('ока') # проверить, кончается ли строка на другую строку (см. startswith())
True
>>> '01\t012\t0123\t01234'.expandtabs(4)
'01  012 0123    01234'
>>> s.find('строка')  # найти первое вхождение подстроки (-1, если не найдена)
9
>>> s.find('о', 9, 15) # найти первое вхождение, начиная с 9 позиции по 15
12
>>> s.index('ОКА') # то же, что find, но бросает исключение, если не найдена
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
>>> # Также есть rfind() и rindex().
...  # Есть методы isalnum(), isalpha(), isdecimal(), isdigit(), islower(), isnumeric()
... # isspace(), isupper() для проверки типа символов
... '123'.isdigit()
True
>>> '12d4'.isdigit()
False
>>> s.split('а') # разбивает строку на список строк разделителем
['Тестов', 'я строк', '']
>>> s.split()    # по умолчанию разделитель — пробельные символы
['Тестовая', 'строка']
>>> '<->'.join(map(str, range(10))) # склеивает строки последовательности разделителем
'0<->1<->2<->3<->4<->5<->6<->7<->8<->9'
>>>
```

Работа со строками (2/2)

```
>>> s = 'Тестовая строка'
>>> s.lower(), s.upper() # привести к нижнему/верхнему регистру
('тестовая строка', 'ТЕСТОВАЯ СТРОКА')
>>> s.partition('ая') # разделить на три части: до разделителя, разделитель, после разд.
('Тестов', 'ая', ' строка')
>>> s.replace('т', '(т)') # заменить подстроку
'Тес(т)овая с(т)рока'
>>> s.center(80)   # выровнять символом по центру (заполнитель по умолчанию: пробел)
'                              Тестовая строка                               '
>>> s.rjust(80, '.') # выровнять по правому краю, ljust() — по левому
'.....................................................................Тестовая строка'
>>> '  test '.strip() # удалить пробельные символы
'test'
>>> # Подробнее: http://docs.python.org/3/library/stdtypes.html#textseq
... pass
>>>
```

Ввод/вывод в Python

```
>>> import io
>>> # open() открывает файл и возвращает "файл-подобный" объект
... f = open("test.txt", "w")  # "w" - создать файл для записи
>>> f.write("Те")              # записать в file-подобный объект
2
>>> f.write("ст!\n")
4
>>> # print() имеет аргумент, позволяющий указать файл-подобный объект для записи:
... print("1 2 3", file=f)
>>> f.close()                  # закрыть файл
>>> f = open("test.txt")       # "r" - открыть для чтения, по умолчанию "r"
>>> f.read(3)                  # прочитать 3 символа из файла
'Тес'
>>> f.readline()               # прочитать одну строку из файла
'т!\n'
>>> f.tell()                   # текущая позиция в файле
10
>>> f.seek(2, io.SEEK_SET)     # переместиться в позицию в файле
2
>>> f.readline()
'ест!\n'
>>> f.read()                   # прочитать до конца файла
'1 2 3\n'
>>> f.close()
>>> f = open("test.txt", "rb") # "b" — открыть в двоичном режиме ("t" для текстового)
>>> f.readline()
b'\xd0\xa2\xd0\xb5\xd1\x81\xd1\x82!\n'
>>> f.close()
>>> # Можно указать кодировку при чтении/записи в текстовом режиме
... f = open("test.txt", encoding="cp1251")
>>> f.read()
'РўРµСЃС‚!\n1 2 3\n'
>>>
```

08.03.2014 Владимир Владимирович Руцкий 11

Стандартные потоки ввода/вывода

```
>>> # В Python по умолчанию есть два файловых потока для ввода с консоли и
... # для вывода в консоль
... import sys
>>> sys.stdout.write("Это вывод\n")
Это вывод
10
>>> t = sys.stdin.read(4)
Это ввод
>>> t
'Это '
>>> sys.stdin.read(4)
'ввод'
>>>
```

Типы ошибок

- Синтаксические ошибки — возникают при загрузке модуля (разборе очередной строки в интерактивном режиме)

```
>>> while True print("test")
  File "<stdin>", line 1
    while True print("test")
                    ^
SyntaxError: invalid syntax
>>>
```

- Фатальные ошибки внутри интерпретатора Python или библиотек

  Crash. «Программа совершила недопустимую операцию и будет закрыта»

- Ошибки времени выполнения — **исключительные ситуации**

```
>>> def div(a, b):
...     return a / b
...
>>> div(1, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in div
ZeroDivisionError: division by zero
>>>
```

Исключительные ситуации

- Популярный механизм обработки ошибок
- Деление на ноль, закончилась память, не удалось открыть файл и др. — **исключительные ситуации**
- Все исключения имеют:
  - **тип** — класс исключения,
  - **объект-исключение** — может хранить дополнительную информацию
- Для обработки исключений используется конструкция:

```
try:
    # БЛОК-КОДА
    ...
except ...: # ОПИСАНИЕ ТИПА ИСКЛЮЧЕНИЯ
    # БЛОК-ОБРАБОТЧИКА-ИСКЛЮЧЕНИЯ
    ...
```

Обработка исключительных ситуаций

```
# блок кода 1
try:
    ... # блок кода 2
except ExceptionType1:
    ... # блок обработки исключения ExceptionType1
except ExceptionType2:
    ... # блок обработки исключения ExceptionType2
# Продолжение блока кода 1
```

Если в *блоке кода 2* происходит необработанная исключительная ситуация, то:

- создаётся объект-исключение (класс + данные), назовём его `exc`
- *блок кода 2* прерывается в месте, где произошло исключение
- среди конструкций `except` последовательно ищется блок с подходящим типом исключения (по типу исключения)
  - если обработчик найден, то выполняется его блок, затем упраление передаётся в продолжение блока кода 1
  - если обработчик не найдён, то исключение не обработано и произойдёт переход к внешнему `try-except` обработчику исключений (если его нет, то программа завершится с ошибкой)

Пример

```
>>> int('123')
123
>>> int('trash')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'trash'
>>>
```

```
while True:
    s = input("Please enter a number: ")
    try:
        x = int(s)
        break  # остановить цикл
    except ValueError:
        print("Can't parse '{0}' as number.".format(s))
    print("Try again")

print("You entered:", x)
```

Конструкция try-except

```
import sys
try:
    # <Блок кода>
    f = open('myfile.txt')
    s = f.readline()
    v = float(s.strip())
    r1 = 1 / int(v)
    r2 = v**v

# <ТипИсключения> as <Объект-Исключение>
except IOError as exc:
    errno, strerror = exc.args
    print("I/O error({0}): {1}".format(errno, strerror))
except ValueError:
    print("Could not convert data to an numer type.")
# Обработчик для нескольких исключений
except (ZeroDivisionError, OverflowError):
    print("Floating point operation exception.")
# Обработчик для всех типов исключений
except:
    # В sys можно найти всю информацию об исключении, включая
    # стек вызовов.
    print("Unexpected error:", sys.exc_info()[0])
    raise   # Оставляем текущее исключение необработанным --- произойдёт поиск
            # обработчика в следующем вложенном try: ... except: ...
# Блок, который выполняется если не произошло никакого (непойманного в
# <блоке кода>) исключения.
else:
    print("No exceptions!")
```

Иерархия исключений

- исключения образуют иерархию по типам
- Обработчик с типом T подходит для исключения с типом `type(exc)`, если
  - `type(exc) == T`, или
  - `type(exc)` *наследуется* от T

Или просто:

```
if isinstance(exc, T):
    ...
```

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      ...
```

Порождение исключений

Вызвать (*породить*, *бросить*) исключение можно явно:

```
raise some_object
```

Пример:

```
>>> try:
...     # Бросаем исключение стандартного типа (можно любого своего).
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst))     # объект-исключение
...     print(inst.args)      # Класс Exception для удобства сохраняет свои
...                             # аргументы в .args
...     print(inst)
...
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
>>>
```

Конструкция `try-finally`

Есть синтаксис для гарантированного выполнения кода:

```
try:
    ... # блок кода 1
except SomeExcType:
    ... # блок кода except
finally:
    ... # блок кода finally
... # продолжение блока кода
```

- Если
  - *блок кода 1* не бросил исключения
  - *блок кода 1* бросил исключение, но оно было обработано в `except`

  то перед выполнением *продолжения блока кода* выполнится *блок кода finally*
- Если
  - *блок кода 1* бросил исключение и оно не было поймано в `except`
  - *блок кода 1* бросил исключение, оно было поймано `except`, но внутри *блока кода except* произошло непойманное исключение

  то перед переходу к внешнему обработчику исключений выполнится *блок кода finally*

Пример finally

```
f = open("test.txt")
try:
   ...       # обработка содержимого файла
finally:
   f.close() # выполнится всегда после завершения блока try
```

```
try:
    main()
finally:
    print("Good bye!")
```

Практика