Язык программирования Python

Цикл **for**. Списки.

Алексей Андреевич Сорокин

спецкурс, ОТИПЛ МГУ,
осенний семестр 2017–2018 учебного года
26 сентября 2017 г.

Оператор **for**

- Число повторов в while задаётся условием.

Оператор **for**

- Число повторов в while задаётся условием.
- Это неудобно, если оно известно заранее.

Оператор **for**

- Число повторов в while задаётся условием.
- Это неудобно, если оно известно заранее.
- Инструкция for позволяет делать ровно k повторов:
  for i **in range**(k):
  ⟨TAB⟩ ⟨операция_1⟩
  ⟨TAB⟩ ...
  ⟨TAB⟩ ⟨операция_m⟩

Оператор **for**

- Число повторов в while задаётся условием.
- Это неудобно, если оно известно заранее.
- Инструкция for позволяет делать ровно k повторов:
  for i **in range**(k):
  ⟨TAB⟩ ⟨операция_1⟩
  ⟨TAB⟩ ...
  ⟨TAB⟩ ⟨операция_m⟩
- **range**(k) возвращает все числа в диапазоне от 0 до k (невключительно), а **for** пробегает по этим числам.

Оператор **for**

- Число повторов в while задаётся условием.
- Это неудобно, если оно известно заранее.
- Инструкция for позволяет делать ровно k повторов:
  for i **in range**(k):
  ⟨TAB⟩ ⟨операция_1⟩
  ⟨TAB⟩ ...
  ⟨TAB⟩ ⟨операция_m⟩
- **range**(k) возвращает все числа в диапазоне от 0 до k (невключительно), а **for** пробегает по этим числам.
- Это только частный случай использования операторов **range** и **for**.

Цикл for

Вычисление степени числа:

```
s = int(input())
t = int(input())
d = 1
for i in range(t):
    d *= s
print("{} in power {} is {}".format(s, t, d))
```

```
c:\Преподавание\ОТИПЛ\Python\2017_осень>python for_example.py
3
2
3 in power 2 is 9

c:\Преподавание\ОТИПЛ\Python\2017_осень>python for_example.py
2
4
2 in power 4 is 16

c:\Преподавание\ОТИПЛ\Python\2017_осень>python for_example.py
5
0
5 in power 0 is 1
```

Списки

- Цикл **for** может пробегать по любому итерируемому объекту (допускающему перебор).

Списки

- Цикл **for** может пробегать по любому итерируемому объекту (допускающему перебор).
- Один из примеров — **списки**.

Списки

- Цикл **for** может пробегать по любому итерируемому объекту (допускающему перебор).
- Один из примеров — **списки**.
- Список — упорядоченный набор элементов (возможно, с повторами).

```
a = [1, 2, 3, 4] # создаём список
for x in a:
    print("Square of {} is {}".format(x, x*x))
```

```
>>> a = [1, 2, 3, 4]
>>> for x in a:
...     print("Square of {} is {}".format(x, x*x))
...
Square of 1 is 1
Square of 2 is 4
Square of 3 is 9
Square of 4 is 16
```

Доступ к элементам списка

- i-ый элемент списка: $a[i]$ (нумерация с 0).

Доступ к элементам списка

- i-ый элемент списка: $a[i]$ (нумерация с 0).
- Можно использовать для перебора:

```
a = [1, 2, 3, 4] # создаём список
for i in range(len(a)):
    print("Square of {}-th element is {}".format(i, a[i]*a[i]))
```

Доступ к элементам списка

- i-ый элемент списка: $a[i]$ (нумерация с 0).
- Можно использовать для перебора:

```
a = [1, 2, 3, 4] # создаём список
for i in range(len(a)):
    print("Square of {}-th element is {}".format(i, a[i]*a[i]))
```

- Можно изменять элементы списка:

```
a[1] = 4 # теперь a = [1, 4, 3, 4]
```

Доступ к элементам списка

- i-ый элемент списка: $a[i]$ (нумерация с 0).
- Можно использовать для перебора:

```
a = [1, 2, 3, 4] # создаём список
for i in range(len(a)):
    print("Square of {}-th element is {}".format(i, a[i]*a[i]))
```

- Можно изменять элементы списка:

```
a[1] = 4 # теперь a = [1, 4, 3, 4]
```

- Индексы можно отсчитывать с конца (нумерация с -1):

```
a[-1] = 5 # теперь a = [1, 4, 5, 4]
```

Доступ к элементам списка

- Доступ по неправильному индексу: **IndexError** (Python не проверяет границы списка):

```
>>> a = [1, 7, 2, 5]
>>> a[4] = 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
>>>
```

Доступ к элементам списка

- Доступ по неправильному индексу: **IndexError** (Python не проверяет границы списка):

```
>>> a = [1, 7, 2, 5]
>>> a[4] = 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
>>>
```

- Длина списка: функция **len**:

```
>>> a = [1, 7, 2, 5]
>>> a[4] = 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
>>>
```

- [] — пустой список.

Доступ к элементам списка

- Доступ по неправильному индексу: **IndexError** (Python не проверяет границы списка):

```
>>> a = [1, 7, 2, 5]
>>> a[4] = 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
>>>
```

- Длина списка: функция **len**:

```
>>> a = [1, 7, 2, 5]
>>> a[4] = 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
>>>
```

- [] — пустой список.
- Элементы списка — произвольные объекты (необязательно однотипные):

```
a = [1, "abc", [2, 3], True]
```

Операции над списками

Операция	Описание
$a = []$	создаёт пустой список
$a = b + c$	конкатенирует списки a и b
$a = b * k$	копирует список b k раз
	возвращает новый список
$a.\mathrm{append}(x)$	добавляет x в конец a
	изменяет a, ничего не возвращает
$a.\mathrm{extend}(b)$	присоединяет список b к a
	изменяет a, ничего не возвращает
x **in** a	проверяет наличие x в a
$x = a.\mathrm{pop}(i)$	возвращает i-ый элемент,
	удаляя его из списка

Операции над списками

```
>>> b = [1, 3]
>>> a = b * 2
>>> a
[1, 3, 1, 3]
>>> x = a.pop(2)
>>> x
1
>>> a
[1, 3, 3]
>>> a.append(x+1)
>>> a
[1, 3, 3, 2]
>>> a.extend([7, 5, 3])
>>> a
[1, 3, 3, 2, 7, 5, 3]
>>> max(a)
7
>>> min(a)
1
```

Возможности **range**

- При последовательном переборе лучше использовать конструкцию for **x** in **a**, чем **range**.

Возможности **range**

- При последовательном переборе лучше использовать конструкцию for **x** in **a**, чем **range**.
- Но **range** полезен в других случаях.
- **Задача**: напечатать все нечётные элементы списка (то есть стоящие на чётных позициях).

Возможности **range**

- При последовательном переборе лучше использовать конструкцию for **x** in **a**, чем **range**.
- Но **range** полезен в других случаях.
- **Задача**: напечатать все нечётные элементы списка (то есть стоящие на чётных позициях).
- **Решение**: использовать **range** с шагом 2.

```
for i in range(0, len(a), 2):
    print(a[i])
```

Возможности **range**

- При последовательном переборе лучше использовать конструкцию for **x** in **a**, чем **range**.
- Но **range** полезен в других случаях.
- **Задача**: напечатать все нечётные элементы списка (то есть стоящие на чётных позициях).
- **Решение**: использовать **range** с шагом 2.

```
for i in range(0, len(a), 2):
    print(a[i])
```

- Синтаксис **range**:

Обозначение	значения
range(n)	$0, 1, \ldots, n-1$
range(m, n)	$m, \ldots, n-1$
range(m, n, k)	$m, m+k, m+2k, \ldots$
range(m, n, -1), $m > n$	$m, m-1, \ldots, n+1$
range(m, n, k), $k < 0$	$m, m-k, m-2k, \ldots$

Срезы

- Функциональность **range** можно имитировать с помощью срезов.

Срезы

- Функциональность **range** можно имитировать с помощью срезов.
- Синтаксис:

Обозначение	значения
$a[i :]$	$[a[i], a[i+1], \ldots, a[-1]]$
$a[: i]$	$[a[0], a[i+1], \ldots, a[i-1]]$
$a[i : j]$	$[a[i], a[i+1], \ldots, a[j-1]]$
$a[i : j : k]$	$[a[i], a[i+k], a[i+2k], \ldots]$
$a[:: -1]$	$[a[-1], a[-2], \ldots, a[0]$
$a[i : j : k],\ k < 0$	$[a[i], a[i-k], a[i-2k], \ldots$

Срезы

- Функциональность **range** можно имитировать с помощью срезов.
- Синтаксис:

Обозначение	значения
$a[i:]$	$[a[i], a[i+1], \ldots, a[-1]]$
$a[:i]$	$[a[0], a[i+1], \ldots, a[i-1]]$
$a[i:j]$	$[a[i], a[i+1], \ldots, a[j-1]]$
$a[i:j:k]$	$[a[i], a[i+k], a[i+2k], \ldots]$
$a[::-1]$	$[a[-1], a[-2], \ldots, a[0]]$
$a[i:j:k],\ k<0$	$[a[i], a[i-k], a[i-2k], \ldots$

- Решение задачи про нечётные элементы:

```
for x in a [::2]:
    print (x)
```

Срезы

```
>>> a = list(range(1, 8))
>>> a
[1, 2, 3, 4, 5, 6, 7]
>>> a[3:]
[4, 5, 6, 7]
>>> a[:4]
[1, 2, 3, 4]
>>> a[3:6]
[4, 5, 6]
>>> a[1:6:2]
[2, 4, 6]
>>> a[5:0:-2]
[6, 4, 2]
>>> a[::-1]
[7, 6, 5, 4, 3, 2, 1]
>>> a[-2::-2]
[6, 4, 2]
```

- Срезы можно использовать для присваивания:

```
>>> a[1::3]
[2, 5]
>>> a[1::3] = [10, 10]
>>> a
[1, 10, 3, 4, 10, 6, 7]
>>>
```

Копирование списков

- При присваивании $\mathrm{a} = \mathrm{b}$ не создаётся новый список *a*, а просто ещё одна ссылка на старый список *b*.
- Это делается, чтобы избежать копирования большого объёма данных.

Копирование списков

- При присваивании $\mathrm{a} = \mathrm{b}$ не создаётся новый список *a*, а просто ещё одна ссылка на старый список *b*.
- Это делается, чтобы избежать копирования большого объёма данных.
- Но тогда любое изменение *a* отражается на *b* и наоборот (они занимают одну и ту же область памяти).

```
>>> a = [1, 2, 3, 4, 5]
>>> b = a
>>> b[0] = 3
>>> a
[3, 2, 3, 4, 5]
```

Копирование списков

- При присваивании $\mathrm{a} = \mathrm{b}$ не создаётся новый список *a*, а просто ещё одна ссылка на старый список *b*.
- Это делается, чтобы избежать копирования большого объёма данных.
- Но тогда любое изменение *a* отражается на *b* и наоборот (они занимают одну и ту же область памяти).

```
>>> a = [1, 2, 3, 4, 5]
>>> b = a
>>> b[0] = 3
>>> a
[3, 2, 3, 4, 5]
```

- Если это нежелательно, лучше использовать срезы.

```
>>> a
[3, 2, 3, 4, 5]
>>> b = a[:]
>>> b[0] = 5
>>> b
[5, 2, 3, 4, 5]
>>> a
[3, 2, 3, 4, 5]
```

Генераторы списков

- Часто нужно применить одну и ту же операцию ко всем элементам списка.
- Например, возвести их в квадрат.

Генераторы списков

- Часто нужно применить одну и ту же операцию ко всем элементам списка.
- Например, возвести их в квадрат.
- Используют так называемые *генераторы списков*:

$$b = [\langle EXPR \rangle \, for \, \mathbf{x} \, in \, \mathbf{a}]$$

- $\langle EXPR \rangle$ — произвольное выражение.

Генераторы списков

- Часто нужно применить одну и ту же операцию ко всем элементам списка.
- Например, возвести их в квадрат.
- Используют так называемые *генераторы списков*:

$$b = [\langle EXPR \rangle \, for \; \mathbf{x} \; in \; \mathbf{a}]$$

- $\langle EXPR \rangle$ — произвольное выражение.
- Например, все квадраты

```
b = [x * x for x in a]
```

Генераторы списков

- Часто нужно применить одну и ту же операцию ко всем элементам списка.
- Например, возвести их в квадрат.
- Используют так называемые *генераторы списков*:

$$b = [\langle EXPR \rangle \, for \; \mathbf{x} \; in \; \mathbf{a}]$$

- $\langle EXPR \rangle$ — произвольное выражение.
- Например, все квадраты

```
b = [x * x for x in a]
```

- Можно вернуть результат применения выражения только к некоторым элементам:
- Квадраты нечётных чисел, входящих в список:

```
b = [x * x for x in a if x % 2 == 1]
```

Вложенные списки

- Элементами списков могут быть другие списки.
- Это позволяет кодировать таблицы, матрицы и т. д.

Вложенные списки

- Элементами списков могут быть другие списки.
- Это позволяет кодировать таблицы, матрицы и т. д.
- Пример: таблица 4×4 для первых 16 чисел:

```
a = [None] * 4 # создаём четыре списка
# None – специальное "нулевое" значение
for i in range(4):
    a[i] = list(range(4 * i, 4 * (i+1)))
```

Вложенные списки

- Элементами списков могут быть другие списки.
- Это позволяет кодировать таблицы, матрицы и т. д.
- Пример: таблица 4×4 для первых 16 чисел:

```
a = [None] * 4 # создаём четыре списка
# None – специальное "нулевое" значение
for i in range(4):
    a[i] = list(range(4 * i, 4 * (i+1)))
```

- Важно: при инициализации лучше не писать $a = [[]] * 4$ — тогда все четыре элемента списка будут ссылаться на одно значение (здесь это неважно, так как всё равно мы присваиваем элементам списков новые значения, а не изменяем их "на месте").

Вложенные списки

- Элементами списков могут быть другие списки.
- Это позволяет кодировать таблицы, матрицы и т. д.
- Пример: таблица 4×4 для первых 16 чисел:

```
a = [None] * 4 # создаём четыре списка
# None – специальное "нулевое" значение
for i in range(4):
    a[i] = list(range(4 * i, 4 * (i+1)))
```

- Важно: при инициализации лучше не писать $a = [[]] * 4$ — тогда все четыре элемента списка будут ссылаться на одно значение (здесь это неважно, так как всё равно мы присваиваем элементам списков новые значения, а не изменяем их "на месте").
- Можно применить генераторы списков:

```
a = [list(range(i * 4, i * 4 + 4)) for i in range(4)]
```

Вложенные списки

- Элементами списков могут быть другие списки.
- Это позволяет кодировать таблицы, матрицы и т. д.
- Пример: таблица 4×4 для первых 16 чисел:

```
a = [None] * 4 # создаём четыре списка
# None – специальное "нулевое" значение
for i in range(4):
    a[i] = list(range(4 * i, 4 * (i+1)))
```

- Важно: при инициализации лучше не писать $a = [[]] * 4$ — тогда все четыре элемента списка будут ссылаться на одно значение (здесь это неважно, так как всё равно мы присваиваем элементам списков новые значения, а не изменяем их "на месте").
- Можно применить генераторы списков:

```
a = [list(range(i * 4, i * 4 + 4)) for i in range(4)]
```

- Задача: поменять столбцы и строки списка местами ("транспонировать матрицу").

Строки

- Строки — последовательности символов. Задаются в двойных или одинарных кавычках.

```
a = "abc"
```

Строки

- Строки — последовательности символов. Задаются в двойных или одинарных кавычках.

```
a = "abc"
```

- Элементы строк — любые UTF-8 символы.

Строки

- Строки — последовательности символов. Задаются в двойных или одинарных кавычках.

  a = "abc"

- Элементы строк — любые UTF-8 символы.
- Строки допускают большинство операций со списками, но есть отличия:

```
>>> a = "abc"
>>> a[1] = "d"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> a = a[:1] + "d" + a[2:]
>>> a
'adc'
```

- Для строк невозможно присваивание по индексу.

Строки

- Строки — последовательности символов. Задаются в двойных или одинарных кавычках.

  a = "abc"

- Элементы строк — любые UTF-8 символы.
- Строки допускают большинство операций со списками, но есть отличия:

```
>>> a = "abc"
>>> a[1] = "d"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> a = a[:1] + "d" + a[2:]
>>> a
'adc'
```

- Для строк невозможно присваивание по индексу.
- Строки — неизменяемые объекты, нельзя изменить существующий объект, можно только создать другой с тем же именем.

Операции над строками

Операция	Описание
$a = ""$	создаёт пустую строку
$a = b + c$	конкатенирует строку a и b
$a = b * k$	копирует строку b k раз
$a = b[i : j : k]$	срез (смысл параметров тот же)
$\mathrm{len}(a)$	длина строки

Операции над строками

Операция	Описание
a.replace(x, y)	заменяет все вхождения подстроки x на y
a **in** x	проверяет вхождение подстроки x
a.find(x)	ищет начало первого вхождения подстроки x (-1 если отсутствует)
a.count(x)	число вхождений x в a
a.lower() (a.upper())	приводит к нижнему (верхнему) регистру
a.isupper() (a.islower())	находится ли в верхнем (нижнем регистре)
a.isdigit()	является ли последовательностью цифр
a.isalpha()	является ли последовательностью букв

Больше функций: `https://docs.python.org/3/library/stdtypes.html#string-methods`.

Операции над строками

```
>>> a = "abcabacd"
>>> a.find("abc")
0
>>> a.count("ab")
2
>>> b = a.replace("ab", "A")
>>> b
'AcAacd'
>>> b.lower()
'acaacd'
>>> b.islower()
False
>>> b.upper()
'ACAACD'
>>> c = "13.2"
>>> c.isdigit()
False
>>> c = c[:2]
>>> c
'13'
>>> c.isdigit()
True
>>> c.isalpha()
False
```

Операции над строками

a.lstrip() (*a.rstrip()*)	удаляет пробельные символы в начале (конце) строки
a.strip()	удаляет пробельные символы по краям строки
a.strip(x)	удаляет подстроку *x* по краям строки
a.split()	разбивает *a* по пробельным символам
a.split(x)	разбивает *a* по подстроке *x*
a.join(b)	соединяет строки в списке *b* через подстроку *a*

```
a = "ab"
b = "cd"
c = "ef"
print("".join([a, b, c])) # будет выведено "abcdef"
```

Операции над строками

```
>>> a = "  abc "
>>> a = a.strip()
>>> a += "aca"
>>> a
'abcaca'
>>> a.strip("a")
'bcac'
>>> a.split("a")
['', 'bc', 'c', '']
>>> a = "Let us split the sentence, we can do it!"
>>> a.split()
['Let', 'us', 'split', 'the', 'sentence,', 'we', 'can', 'do', 'it!']
>>> "_".join(a.split())
'Let_us_split_the_sentence,_we_can_do_it!'
```

Операции над строками

```
>>> a = "  abc "
>>> a = a.strip()
>>> a += "aca"
>>> a
'abcaca'
>>> a.strip("a")
'bcac'
>>> a.split("a")
['', 'bc', 'c', '']
>>> a = "Let us split the sentence, we can do it!"
>>> a.split()
['Let', 'us', 'split', 'the', 'sentence,', 'we', 'can', 'do', 'it!']
>>> "_".join(a.split())
'Let_us_split_the_sentence,_we_can_do_it!'
```

- strip() считает последовательность пробельным символов за один символ.
- strip(x), напротив, не соединяет одинаковые разделители.