Лекция 1 "Введение в программирование на Python"

часть 2

Финансовый университет при Правительстве РФ, лектор С.В. Макрушин

v 0.8 31.08.2021

Разделы:

-

- к оглавлению

Типы данных

- к оглавлению

Все данные в языке Python представлены объектами. Каждый объект имеет:

- уникальный идентификатор (id)
- тип данных
- значение (изменяемое или неизменяемое)

Основные числовые типы данных и операции над ними

- к оглавлению

Тип *int* - целые числа. Размер целого числа в Python ограничен лишь объемом оперативной памяти.

In [2]:

```
# целое число:
2147483647
```

Out[2]:

```
2147483647
```

In [3]:

```
# тип целого числа:
type(2147483647)
```

Out[3]:

```
int
```

In [4]:

```
# очень большое целое число:
99999999999999999999999
```

Out[4]:

```
99999999999999999999999
```

In [5]:

```
# тип очень большого целого числа:
type(99999999999999999999999)
```

Out[5]:

```
int
```

In [6]:

```
0b10 # целое число, заданное в двоичном формате
```

Out[6]:

```
2
```

In [7]:

```
0o17 # целое число, заданное в восьмеричном формате
```

Out[7]:

```
15
```

In [149]:

```
0x10 # целое число, заданное в шестнадцатеричном формате
```

Out[149]:

```
16
```

In [8]:

```
0xA
```

Out[8]:

```
10
```

In [9]:

```
0xF
```

Out[9]:

```
15
```

Тип *float* - вещественные числа. Используются числа с плавающей точкой двойной точности.

In [11]:

```
# вещественное число:
42.1
```

Out[11]:

```
42.1
```

In [12]:

```
# тип вещественного числа:
type(42.1)
```

Out[12]:

```
float
```

In [13]:

```
# задание вещественного числа в экспоненциальной форме:
4.21e+1
```

Out[13]:

```
42.1
```

In [14]:

```
# задание вещественного числа в экспоненциальной форме:
4.21e-1
```

Out[14]:

```
0.421
```

In [15]:

```
# задание целого числа в виде вещественного числа:
42.0
```

Out[15]:

```
42.0
```

In [16]:

```
type(42.0)
```

Out[16]:

```
float
```

Тип *complex* - комплексные числа.

In [17]:

```
# комплексное число:
2+2.1j
```

Out[17]:

```
(2+2.1j)
```

In [18]:

```
# тип комплексного числа:
type(2+2.1j)
```

Out[18]:

```
complex
```

В Python элементарные числовые типы, такие как *int* , *float* , *complex* , а также строковой тип *str* , логический тип *bool* и некоторые другие типы **являются неизменяемыми**. Это означает, что однажды установив значение объекта (не переменной!) этого типа, его уже нельзя будет изменить.

Далее эта специфика Python будет рассмотрена подробнее.

Математические операции над числовыми типами данных

- к оглавлению

Операторы позволяют произвести определенные действия с данными.

\+ - сложение:

In [19]:

```
10 + 5 # Целые числа
```

Out[19]:

```
15
```

In [20]:

```
12.4 + 5.6 # Вещественные числа
```

Out[20]:

```
18.0
```

In [21]:

```
10 + 12.4 # Целые и вещественные числа
```

Out[21]:

```
22.4
```

\- - вычитание:

In [22]:

```
10 - 5 # Целые числа
```

Out[22]:

```
5
```

In [23]:

```
12.4 - 5.2 # Вещественные числа
```

Out[23]:

```
7.2
```

In [24]:

```
12 - 5.2 # Целые и вещественные числа
```

Out[24]:

```
6.8
```

* - умножение:

In [25]:

```
10 * 5 # Целые числа
```

Out[25]:

```
50
```

In [26]:

```
12.4 * 5.2 # Вещественные числа
```

Out[26]:

```
64.48
```

In [27]:

```
10 * 5.2 # Целые и вещественные числа
```

Out[27]:

```
52.0
```

/ - деление. **Результатом деления всегда является вещественное число**, даже если производится деление целых чисел.

In [168]:

```
10 / 5 # Деление целых чисел без остатка
```

Out[168]:

```
2.0
```

In [169]:

```
type(10 / 5) # Тип результата деления целых чисел без остатка
```

Out[169]:

```
float
```

In [170]:

```
10 / 3 # Деление целых чисел с остатком
```

Out[170]:

```
3.3333333333333335
```

In [171]:

```
10.0 / 5.0 # Деление вещественных чисел
```

Out[171]:

```
2.0
```

In [172]:

```
type(10.0 / 5.0) # Тип результата деления вещественных чисел без остатка
```

Out[172]:

```
float
```

In [173]:

```
10.0 / 3.0 # Деление вещественных чисел
```

Out[173]:

```
3.3333333333333335
```

In [174]:

```
10 / 5.0 # Деление целого числа на вещественное
```

Out[174]:

```
2.0
```

`//` - деление с округлением вниз. Вне зависимости от типа чисел остаток отбрасывается.

In [175]:

```
 10 // 5 #Деление целых чисел без остатка
```

Out[175]:

```
2
```

In [176]:

```
type(10 // 5) # Тип результата деления целых чисел без остатка
```

Out[176]:

```
int
```

In [177]:

```
10 // 3 # Деление целых чисел с остатком
```

Out[177]:

```
3
```

In [178]:

```
10.0 // 5.0 # Деление вещественных чисел
```

Out[178]:

```
2.0
```

In [179]:

```
type(10.0 // 5.0) # Тип результата деления вещественных чисел
```

Out[179]:

```
float
```

In [181]:

```
10.0 // 3.0 # Деление вещественных чисел
```

Out[181]:

```
3.0
```

In [182]:

```
10 // 3.0 # Деление целого числа на вещественное
```

Out[182]:

```
3.0
```

`%` - остаток от деления:

In [183]:

```
10 % 5 # Остаток от деление целых чисел
```

Out[183]:

```
0
```

In [184]:

```
10 % 3 # Остаток от деление целых чисел
```

Out[184]:

```
1
```

In [185]:

```
10.0 % 3.0 # Остаток от деление вещественных чисел
```

Out[185]:

```
1.0
```

In [188]:

```
10 % 3.0 # Остаток от деление вещественных чисел
```

Out[188]:

```
1.0
```

** - возведение в степень:

In [189]:

```
10 ** 2
```

Out[189]:

```
100
```

In [190]:

```
10.0 ** 2
```

Out[190]:

```
100.0
```

унарный - (минус) и унарный + (плюс):

In [28]:

```
 -(-10)
```

Out[28]:

```
10
```

In [29]:

```
-(-10.0)
```

Out[29]:

```
10.0
```

Операции над числами разных типов возвращают число, имеющее более сложный тип из типов, участвующих в операции. Приоритеты типов:

1. целые числа (самый простой тип);
2. вещественные числа;
3. комплексные числа (самый сложный тип).

При вычислении математических выражений действует общепринятый приоритет выполнения операций:

In [30]:

```
3*2 + 1 # операции с более низким приоритетом принято выделять пробелами
```

Out[30]:

```
7
```

>

Преобразование типов

Для преобразования элемента данных из одного типа в другой мы можем использовать конструкцию *datatype(item)*.

При выполнении преобразования типа создается новый объект с необходимым нам типом и значением, соответствующим значению объекта другого типа.

In [31]:
```
42
```

Out[31]:
```
42
```

In [32]:
```
type(42)
```

Out[32]:
```
int
```

In [33]:
```
float(42)
```

Out[33]:
```
42.0
```

In [34]:
```
type(float(42))
```

Out[34]:
```
float
```

In [35]:
```
42.1
```

Out[35]:
```
42.1
```

In [36]:

```
int(42.1)
```

Out[36]:

```
42
```

In [37]:

```
int('42')
```

Out[37]:

```
42
```

In [38]:

```
str(42)
```

Out[38]:

```
'42'
```

Если преобразование невозможно, то возбуждается исключение:

In [39]:

```
int('71s')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-39-4069a33679db> in <module>
----> 1 int('71s')

ValueError: invalid literal for int() with base 10: '71s'
```

Язык Python имеет **сильную типизацию**. Т.е. не позволяет смешивать в выражениях различные типы и не выполняет автоматические неявные преобразования (кроме некоторых особых случаев).

- Основной особый случай: арифметические выражения с различными типами чисел (int, float, complex) - тут производится автоматическое преобразование к наиболее сложному типу из участвующих в выражении.

In [40]:

```
10 + '5'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-40-17f4ad3de8c5> in <module>
----> 1 10 + '5'

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Для таких случаев необходимо использование явного преобразования типов:

In [41]:

```
10 + int('5')
```

Out[41]:

15

>

Переменные

- к оглавлению

Для доступа к объекту предназначены переменные. **В языке Python все переменные являются ссылками на объекты** : при инициализаци в переменной сохраняется ссылка (адрес объекта в памяти компьютера) на объект, благодаря этой ссылке хранящейся в переменной в дальнейшем можно использовать объект из программы.

В Python для присвоения используется = (для сравнения значений на равенство используется ==).

- Первое присвоение значения переменной создает ее.
- Нет необходимости декларировать тип переменной (он опредляется автоматичиески - по типу присваиваемого объекта).

In [3]:

```
a = 1
```

Когда интерпретатор Python выполняет эту инструкцию, он создает объект типа *int* со значением 1, а затем создает ссылку на объект с именем `a` , которая ссылается на только что созданный объект.

Явная и неявная типизация

- В языках программирования с явной типизаций тип новых переменных / функций (их аргументов и возвращаемых значений) *нужно задавать явно*.

Достоинства **явной** типизации:

- Наличие у каждой функции сигнатуры (например int add(int, int)) позволяет без проблем определить, что функция делает.
- Программист сразу записывает, какого типа значения могут храниться в конкретной переменной, что снимает необходимость запоминать это.

Достоинства **неявной** типизации

- Сокращение записи — def add(x, y) короче, чем int add(int x, int y).
- Устойчивость к изменениям. Например если в функции временная переменная была того же типа, что и входной аргумент, то в явно типизированном языке при изменении типа входного аргумента нужно будет изменить еще и тип временной переменной.

В Python используется неявная типизация.

In [4]:

```
print(a)
```

```
1
```

In [5]:

```
a
```

Out[5]:

```
1
```

In [6]:

```
type(a)
```

Out[6]:

```
int
```

In [7]:

```
id(a) # получение уникального идентификатора объекта
```

Out[7]:

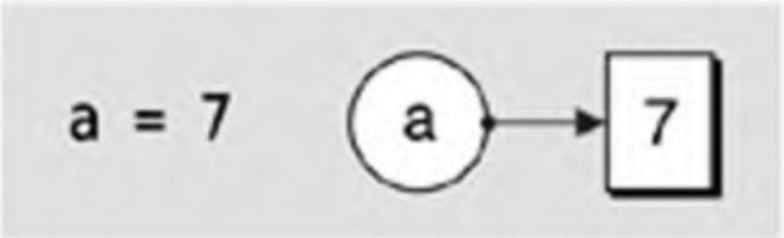
```
140714199523728
```

In [8]:

```
?id
```

In [17]:

```
# Переменные и ссылки на объекты# "присвоение":
a = 7 # вместо изменения значения хранящегося в a, а присваивается ссылка на новый объект (
```



Переменные и ссылки на объекты

In [11]:

```
id(a)
```

Out[11]:

```
140714199523920
```

При работе с объектами, имеющими тип, не допускающий изменения содержимого (например `int`, `float` или `str`), работа со ссылками не отличима от работы непосредственно со значениями. Различия начинают проявляться, когда дело доходит до изменяемых объектов.

In [12]:

```
# "присвоение":
b = a # в новую переменную с именем b записывается ссылка на тот же объект
```

In [13]:

```
# проверяем:
id(b) # переменная b ссылается на тот же объект, что и переменная a
```

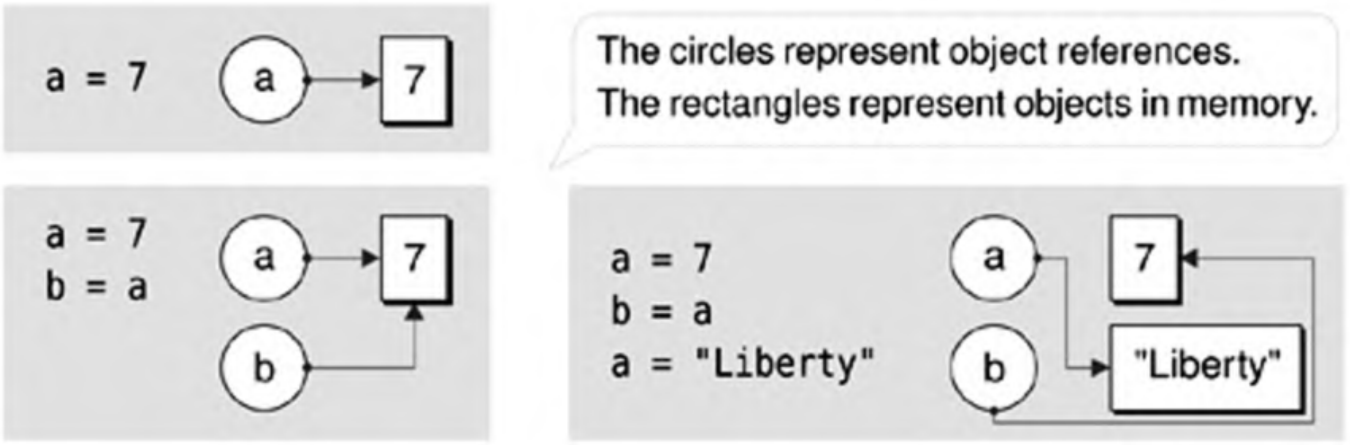
Out[13]:

```
140714199523920
```

Оператор `=` - это не оператор присваивания значения переменной, как в некоторых других языках программирования.

Оператор `=` связывает ссылку на объект (переменную) с объектом, находящимся в памяти.

- Если ссылка на объект (переменная) уже существует, ее легко можно связать с другим объектом, указав этот объект справа от оператора `=`.
- Если ссылка на объект еще не существует, она будет создана оператором `=`.



Объекты и ссылки на объекты

In [14]:

```
# a = 'Liberty'
```

In [15]:

```
type(a)
```

Out[15]:

```
str
```

Статическая и динамическая типизация

- к оглавлению

-

- **Статическая типизация** - переменная, параметр подпрограммы, возвращаемое значение функции связываются с типом *в момент объявления* , и их тип *не может быть изменен* позже.

Достоинства:

- хороша для написания сложного, но быстрого кода;
- хорошо работает автодополнение в IDE;
- многие ошибки исключаются на стадии компиляции.

Недостатки:

- многословный код;
- сложность написания обобщенных алгоритмов и универсальных коллекций;
- сложности с работой с данными из внешних источников.

В языке Python используется динамическая типизация, т.е. "тип переменной" (являющейся по сути ссылкой на объект) может меняться во время ее жизни.

Примеры кода на разных языках программирования:

```
/* код на C: явная статическая типизация */
unsigned int find_int (int requiered_element, int array[], unsigned int size) {
    for (unsigned int i = 0; i < size; i++) {
        if (requiered_element == array[i]) {
            return i;
        }
    }
    return -1;
}

// код на C++: явная статическая типизация, обобщенный алгоритм
template <class T>
unsigned int find_int (T requiered_element, std::vector<T> array) {
    for (unsigned int i = 0; i < array.size(); i++) {
        if (requiered_element == array[i]) {
            return i;
        }
    }
    return -1;
}
```

```
    # код на Python: неявная динамическая типизация
    def find(requiered_element, lst):
        for (index, element) in enumerate(lst):
            if element == requiered_element:
                return index
        return -1
```

In [18]:

```
type(a) # определяется тип объекта, на который указывает ссылка
```

Out[18]:

```
int
```

In [19]:

```
a = 'Liberty' # связываем ссылку с другим объектом (имеющим другой тип)
```

In [20]:

```
type(a) # определяется тип объекта, на который указывает ссылка
```

Out[20]:

```
str
```

In [21]:

```
# значение переменной b не изменилось:
b, type(b), id(b)
```

Out[21]:

```
(7, int, 140714199523920)
```

Работа с переменными

- к оглавлению

Рассмотрим, как хранение ссылок в переменных влияет на работу с изменяемыми объектами, например списками (динамическими массивами).

In [22]:

```
li1 = [1, 2] # связываем переменную li1 с новым списком, хранящим два значения
li2 = li1
id(li1), id(li2) # обе переменные ссылаются на один объект
```

Out[22]:

```
(1931241327752, 1931241327752)
```

In [23]:

```
li1.append(3) # меняем список, используя переменную li1
li1
```

Out[23]:

```
[1, 2, 3]
```

In [24]:

```
li2 # состояние объекта, на которой ссылается li2, тоже изменилось!
# С неизменяемыми типами (bool, int, float, str) так сделать нельзя!
```

Out[24]:

```
[1, 2, 3]
```

In [25]:

```
id(li1), id(li2) # обе переменные продолжают ссылаться на тот же объект
```

Out[25]:

```
(1931241327752, 1931241327752)
```

In [26]:

```
# Для связывания нескольких переменных (ссылок) с одним объектом
# можно использовать следующий синтаксис:
a1 = a2 = a3 = 13
```

In [27]:

```
a1
```

Out[27]:

```
13
```

In [28]:

```
a3
```

Out[28]:

```
13
```

In [29]:

```
id(a1) == id(a3)
```

Out[29]:

```
True
```

Проверить, ссылаются ли две переменные на один и тот же объект в памяти, позволяет оператор `is` :

In [30]:

```
a1 is a3
```

Out[30]:

```
True
```

In [31]:

```
a1 == a3
```

Out[31]:

```
True
```

In [32]:

```
id(a3)
```

Out[32]:

```
140714199524112
```

In [33]:

```
a3 = a3 + 2 # изменили значение в a3.
a3, id(a3) # мы НЕ изменили значение целочисленного объекта 3, а установили ссылку на новый
```

Out[33]:

```
(15, 140714199524176)
```

In [34]:

```
a1, a2, a3 # в отличие от списков значение изменилось только у  a3
```

Out[34]:

```
(13, 13, 15)
```

В целях эффективности кода интерпретатор производит кэширование малых целых чисел и небольших строк. Это означает, что независимо связанные ссылки будет ссылаться на один и тот же объект.

In [35]:

```
a4 = 13 # механизм кэширования выдает ссылку на уже существующий объек
```

In [36]:

```
a1, a4
```

Out[36]:

```
(13, 13)
```

In [37]:

```
a1 is a4
```

Out[37]:

```
True
```

In [38]:

```
id(a1), id(a4)
```

Out[38]:

```
(140714199524112, 140714199524112)
```

In [39]:

```
z1 = 123812048374719357913758145790132791379481923 81
```

In [40]:

```
z2 = 123812048374719357913758145790132791379481923 81 # для больших чисел механизм кэширован
```

In [41]:

```
z1 is z2 # две переменные ссылаются на разные объекты
```

Out[41]:

```
False
```

In [42]:

```
z1 == z2 # значения различных объектов совпадают
```

Out[42]:

```
True
```

Управление памятью и сборка мусора

- к оглавлению

Интерпретатор Python постоянно отслеживает,сколько переменных ссылается на данный объект:

In [43]:

```
import sys # Подключаем модуль sys
sys.getrefcount(z1) # определяем количество ссылок на объект
```

Out[43]:

```
2
```

In [44]:

```
sys.getrefcount(z2)
```

Out[44]:

2

In [45]:

```
z3 = z1 # создаем еще одну ссылку
```

In [46]:

```
sys.getrefcount(z1) # количество ссылок увеличилось
```

Out[46]:

3

In [47]:

```
sys.getrefcount(z2)  # количество ссылок не изменилось
```

Out[47]:

2

Когда число ссылок /на объект становится равно нулю, объект автоматически удаляется из оперативной памяти. Так работает механизм автоматической сборки мусора в Python.

Исключением являются объекты, которые подлежат кэшированию.

Удалить переменную (т.е. удалить ссылку на объект) можно с помощью инструкции *del* :

In [48]:

```
del z3
```

In [49]:

```
sys.getrefcount(z1)  # количество ссылок уменьшилось
```

Out[49]:

2

Управление памятью и сборка мусора

Ручное управление памятью

- Для создания объекта в динамической памяти программист явно вызывает команду выделения памяти. Эта команда возвращает указатель на выделенную область памяти, который сохраняется и используется для доступа к ней. До тех пор, пока созданный объект нужен для работы программы, программа обращается к нему через ранее сохранённый указатель.

- Когда надобность в объекте проходит, программист явно вызывает команду освобождения памяти, передавая ей указатель на удаляемый объект.
- В любом языке, допускающем создание объектов в динамической памяти, потенциально возможны две проблемы:
  - висячие ссылки
  - утечки памяти.

Сборка мусора

- В системе со сборкой мусора (garbage collection, GC) обязанность освобождения памяти от объектов, которые больше не используются, возлагается на среду исполнения программы. Программист лишь создаёт динамические объекты и пользуется ими, он может не заботиться об удалении объектов. Для осуществления сборки мусора в состав среды исполнения включается специальный программный модуль, называемый "сборщиком мусора". Этот модуль периодически запускается, определяет, какие из созданных в динамической памяти объектов более не используются, и освобождает занимаемую ими память.
- Сборка мусора — технология, позволяющая, с одной стороны, упростить программирование, избавив программиста от необходимости вручную удалять объекты, созданные в динамической памяти, с другой — устранить ошибки, вызванные неправильным ручным управлением памятью (но не другими ресурсами).

В Python выполняется автоматическая сборка мусора.

Именование переменных

- к оглавлению

Каждая переменная должна иметь уникальное имя, состоящее из латинских букв, цифр и знаков подчеркивания, причем имя переменной не может начинаться с цифры.

In [251]:

```
a = 1
a1 = 2
a_b_c = 3
```

In [252]:

```
1a = 1
```

```
  File "<ipython-input-252-cc67e5ecf289>", line 1
    1a = 1
     ^
SyntaxError: invalid syntax
```

При указании имени переменной важно учитывать регистр букв:

In [50]:

```
x = 1
X = 2
```

In [51]:

```
x
```

Out[51]:

```
1
```

In [52]:

```
X
```

Out[52]:

```
2
```

In [53]:

```
abC = 1
abc = 2
abc, abC
```

Out[53]:

```
(2, 1)
```

Сдедует избегать указания символа подчеркивания в начале имени, т. к. идентификаторы с таким символом имеют специальное значение.

В качестве имени переменной нельзя использовать ключевые слова. Документация: Python keywords (http://docs.python.org/3/reference/lexical_analysis.html#keywords)

In [257]:

```
# получение списка всех ключевых слов:
import keyword
keyword.kwlist
```

Out[257]:

```
['False',
 'None',
 'True',
 'and',
 'as',
 'assert',
 'break',
 'class',
 'continue',
 'def',
 'del',
 'elif',
 'else',
 'except',
 'finally',
 'for',
 'from',
 'global',
 'if',
 'import',
 'in',
 'is',
 'lambda',
 'nonlocal',
 'not',
 'or',
 'pass',
 'raise',
 'return',
 'try',
 'while',
 'with',
 'yield']
```

In [54]:

```
with = 1
```

```
  File "<ipython-input-54-fca67dd73332>", line 1
    with = 1
         ^
SyntaxError: invalid syntax
```

Помимо ключевых слов следует избегать совпадений со встроенными идентификаторами. В отличие от ключевых слов, встроенные идентификаторы можно переопределять, но дальнейший результат может быть нежелательным.

In [55]:

```
# проверка идентификатора (работает только в IPython):
?h
```

```
Object `h` not found.
```

In [260]:

```
?abs
```

In [261]:

```
?a
```

Предпочтительный стиль именования переменных. Имена переменных должны состоять из маленьких букв, а слова разделяться символами подчеркивания.

In [77]:

```
my_name = 1
```

In [78]:

```
very_long_name = 2
```

In [56]:

```
for _ in range(5):
    prAint('Hello!')
```

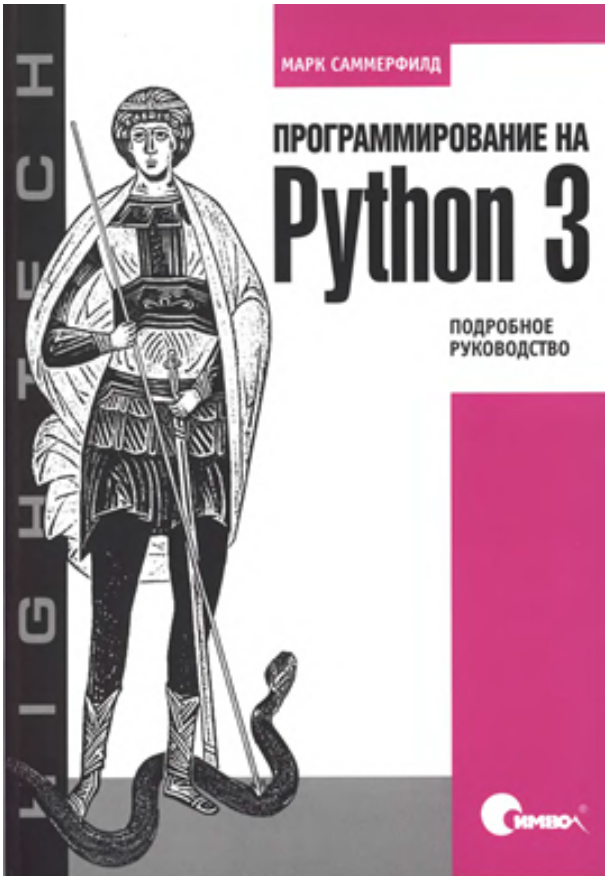
```
Hello!
Hello!
Hello!
Hello!
Hello!
```

>

Задание к следующей лекции

- к оглавлению

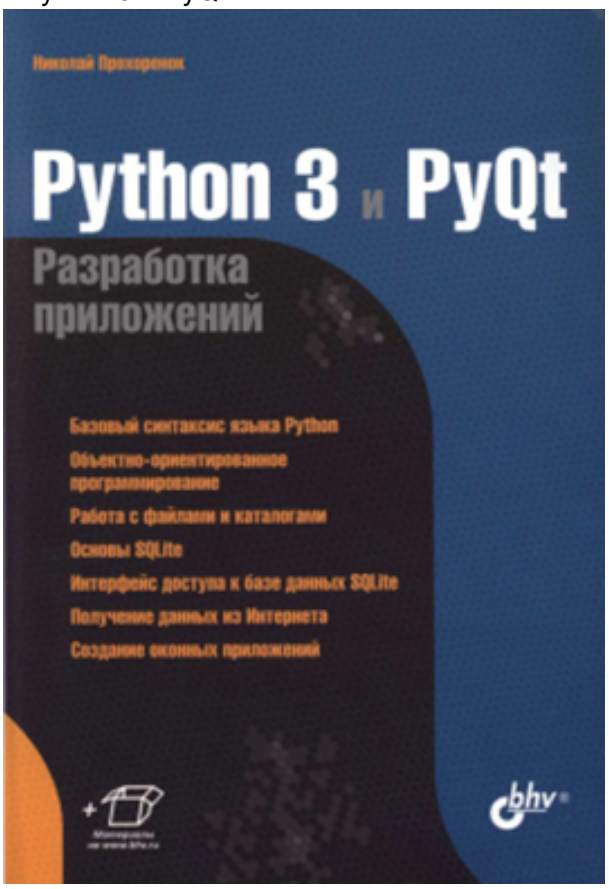
По книге Марка Саммерфильда "Программирование на Python 3"



Книга Марка Саммерфильда

- Гл. 1. Быстрое введение в процедурное программирование (по желанию)
- Гл. 2. Типы данных
- Гл. 3. Типы коллекций (кроме Множества и Отображения)
- Гл. 4. Управляющие структуры и функции (только первый раздел: Управляющие структуры)

По книге Николая Прохоренка "Python 3 и PyQt"



Книга Николая Прохоренка

- Гл. 1. Первые шаги (по желанию)
- Гл. 2. Переменные
- Гл. 3. Операторы
- Гл. 4. Числа
- Гл. 5. Строки
- Гл. 6. Списки и кортежи

In [1]:

```
# загружаем стиль для оформления презентации
from IPython.display import HTML
from urllib.request import urlopen
html = urlopen("file:./lec_v1.css")
HTML(html.read().decode('utf-8'))
```

Out[1]: