Блокнот №2. Информационные технологии.

Списки и операции с ними

Список — это такая структура данных, которая содержит в себе сразу много элементов.

In [ ]:

```
numbers = [4, 8, 9, 2, 6]
```

In [ ]:

```
numbers
```

В списках можно хранить не только числа. Например, создадим список из строк.

In [ ]:

```
strings = ["Hello", "World", "Test"]
```

В списке могут храниться данные разных типов. Например, строки, целые числа, числа с плавающей точкой.

In [ ]:

```
mixed_list = ["Hello", 6, 7.8]
```

Можно обращаться к отдельным элементам списка и работать с ними как с обычными переменными. Чтобы выбрать элемент нужно указать его номер.

In [ ]:

```
print(numbers)
print(numbers[1])
```

Нумерация начинается с нуля!

> На самом деле, у этого правила есть свои рациональные обоснования (http://python-history.blogspot.ru/2013/10/why-python-uses-0-based-indexing.html).

Если в списке есть элементы разных типов, они никак не «мешают» друг другу. Например, наличие в списке строк не превращает другие элементы этого списка в строки.

> Это касается только обычных списков Python. Несколько позже мы будем проходить массивы `numpy` и там всё не так.

In [ ]:

```
print(mixed_list)
print(mixed_list[2]+4)
```

Элементы списка можно менять так же, как значения обычных переменных.

In [ ]:

```
numbers
```

In [ ]:

```
numbers[1]=222
numbers
```

> Списки Python основаны на стандартных C'шных массивах и обладают их свойствами с точки зрения производительности: в частности, обращение к элементу по его индексу имеет сложность $O(1)$, то есть не является массовой операцией.

Чтобы узнать длину списка, можно использовать функцю `len` .

In [ ]:

```
len(numbers)
```

Заметим, что это не индекс последнего элемента, а именно число элементов. Если вам нужно получить последний элемент, то его индексом будет `len(numbers)-1` . Но в Python можно обращаться к элементам списка, считая их «с конца», гораздо проще:

In [ ]:

```
numbers = [4, 8, 2, 5]
numbers[-1]
```

In [ ]:

```
numbers[-2]
```

А вот если вы попытаетесь обратиться к элементу с несуществующим индексом, то получите ошибку.

In [ ]:

```
numbers[5] = 100
```

Однако дописывать элементы в конец можно:

In [ ]:

```
numbers = [7, 6, 2]
print(numbers)
numbers.append(777)
print(numbers)
```

Слово `append` — это так называемый «метод» — функция, «принадлежащая» некоторому объекту (в данном случае — объекту `numbers` типа `list` (список)), и что-то делающая с этим объектом. У `numbers`, как у любого списка, есть много методов. Можно набрать `numbers.`, нажать табуляцию (после точки), и получить список доступных методов. А ещё можно набрать `help(list)` или даже `help(numbers)` (в нашем случае) и получить краткое описание этих методов. Например, так можно узнать, что помимо `append` у списков есть метод `extend`.

In [ ]:

```
print(numbers)
numbers.extend([3, 7, 5])
print(numbers)
```

Метод `extend` позволяет приписать к списку сразу несколько элементов. Он получает на вход список, который нужно приписать: обратите внимание на квадратные скобки внутри круглых при вызове этого метода — они создают новый список, который и передаётся функции `extend`. Методы `append` и `extend` меняют список, к которому они применяются. Иногда вместо этого нужно создать новый список, объединив (конкатенировав!) два других. Это тоже можно сделать.

In [ ]:

```
first_list = [5, 8, 2]
second_list = [1, 9, 4]
new_list = first_list + second_list
print(new_list)
```

Плюсик в данном случае обозначает не поэлементное сложение (как вы могли подумать), а конкатенацию. Списки `first_list` и `second_list` при этом не изменились

In [ ]:

```
print(first_list)
print(second_list)
```

У вас могло возникнуть желание использовать сложение вместо операции `extend`.

In [ ]:

```
print(numbers)
# не надо так
numbers = numbers + [2, 6, 9]
print(numbers)
```

Вообще говоря, этот код сработал, но делать так не следует: при выполнении операции конкатенации создаётся новый список, затем в него копируются все элементы из `numbers`, потом к ним приписываются элементы из второго списка, после чего старый `numbers` забывается. Если бы в

`numbers` было много элементов, их копирование в новый список заняло бы много времени. Гораздо быстрее приписать элементы к уже готовому списку.

Срезы

Иногда нам нужен не весь список, а его кусочек. Его можно получить, указав в квадратных скобках не одно число, а два, разделённых двоеточием.

In [ ]:

```
print(numbers)
print(numbers[1:4])
```

Это называется *slice* (по-русски часто говорят *срез*). Обратите внимание: левый конец среза включается (элемент с индексом 1 — это шестёрка), а правый — нет. Так будет всегда. Это соглашение оказывается удобным, например, потому что позволяет посчитать число элементов в срезе — нужно из правого конца вычесть левый (в данном случае 4-1=3).

Если левый элемент не указан, то он считается началом списка, а если правый — то концом.

In [ ]:

```
print(numbers[7:])
print(numbers[:7])
```

Всегда верно следующее: список `numbers` это то же самое, что `numbers[:k]+numbers[k:]` , где `k` — любой индекс.

Срезы можно использовать для присваивания.

In [ ]:

```
numbers = [5, 8, 9, 10]
print(numbers[1:3])
numbers[1:3]= [55, 77]
print(numbers)
```

Не обязательно, чтобы список, который мы присваиваем срезу, имел ту же длину, что и срез. Можно присвоить более длинный список (тогда исходный список расширится), а можно менее длинный (тогда сузится). Можно использовать срезы, чтобы вставить несколько элементов внутрь списка. (Для одного элемента это можно делать с помощью метода `insert` .)

In [ ]:

```
numbers = [6, 8, 9]
print(numbers[1:1])
# это пустой срез

numbers[1:1] = [99, 77, 55]
print(numbers)
```

> Чтобы вставить какие-то элементы внутрь списка, необходимо освободить для него место, сдвинув все последующие элементы вперёд. Python сделает это автоматически, но время это займёт. Поэтому, по возможности, следует этого избегать, особенно если вы работаете с большими массивами данных. Если вам очень нужно записывать что-нибудь в начало и конец списка, посмотрите на двустороннюю очередь (deque) из модуля `collections` .

Можно удалять элементы списка (и вообще что угодно) или срезы с помощью команды `del` .

In [ ]:

```
numbers = [6, 7, 9, 12, 8, 3]
del(numbers[4]) # удалим 8
print(numbers)
del(numbers[0:2])
print(numbers)
```

Присвоение и копирование списков

Списки могут быть коварными. Пока вы не разберётесь с содержанием этого раздела, ваши программы будут вести себя непредсказуемым образом и вы потратите много времени на их отладку. Так что сейчас самое время сосредоточиться.

In [ ]:

```
first_list = [5, 8, 9, 'Hello']
second_list = first_list
```

In [ ]:

```
first_list
```

In [ ]:

```
second_list
```

Так мы создали два одинаковым списка. Изменим теперь один из них:

In [ ]:

```
second_list[0] = 777
second_list
```

Что вы ожидаете увидеть в `first_list` ?

In [ ]:

```
first_list
```

В `first_list` хранится не сам список, а указатель (ссылка) на него. Когда мы присваиваем значение `first_list` новой переменной `second_list` , мы не производим копирование списка, мы копируем только указатель. То есть `second_list` просто стала другим именем для того же самого списка, что и

`first_list` . Поэтому изменение элементов `second_list` приведет к изменению `first_list` , и наоборот.

Чтобы разобраться в происходящем более подробно, посмотрим, что происходит с нашим кодом строчка за строчкой. Для этого можно использовать сервис Python Tutor (http://pythontutor.com/), с помощью которого можно визуализировать выполнение кода. (Вы можете использовать этот сайт для отладки своих программ.)

In [ ]:

```
%load_ext tutormagic
# Это магия, позволяющая вставить визуализацию с pythontutor прямо в этот notebook.
# Чтобы его использовать, необходимо установить пакет tutormagic
# pip install tutormagic
```

In [ ]:

```
%%tutor --lang python3 # магия
first_list = [5, 8, 9, 'Hello']
second_list = first_list
second_list[0] = 777
```

Если мы хотим создать действительно новый список, то есть *скопировать* существующий, нужно использовать метод `copy()` .

In [ ]:

```
first_list = [6, 9, 2, 5]
third_list = first_list.copy()
print(third_list)
third_list[0] = 100
print(third_list)
print(first_list)
```

Как видите, теперь `first_list` и `third_list` ведут себя независимо. Этот код тоже можно визуализировать.

In [ ]:

```
%%tutor --lang python3
first_list = [6, 9, 2, 5]
third_list = first_list.copy()
print(third_list)
third_list[0] = 100
print(third_list)
print(first_list)
```

Вы также можете встретиться с таким синтаксисом для копирования списков:

In [ ]:

```
first_list = [6, 9, 2, 5]
other_list = first_list[:]
```

Он тоже сработает (по крайней мере, для обычных списков). Здесь `[:]` — это срез, начало которого совпадает с началом исходного списка, а конец — с концом. Такой код вы часто можете встретить в

программах, написанных на Python 2, потому что там не было метода `copy()`.

Цикл for

In [ ]:

```
numbers = [4, 9, 1, 5]
for x in numbers:
    y = x + 1
    print(y)
print("Вот и всё")
print(numbers)
```

Отступами выделено *тело цикла*, в него входят в данном случае две команды. Иными словами, код выше эквивалентен вот такому коду:

Можно воспользоваться Python Tutor и визуализировать этот процесс.

In [ ]:

```
%%tutor --lang python3
numbers = [4, 9, 1, 5]
for x in numbers:
    y = x + 1
    print(y)
print("Вот и всё")
print(numbers)
```

Иногда у нас нет никакого конкретного списка, а просто нужно выполнить некоторый код несколько раз, причём заранее неизвестно, сколько. Для решения этой задачи служит объект `range()`.

In [ ]:

```
for i in range(5):
    print("Hello, i =", i)
```

`range(n)` ведёт себя как как список, содержащий целые числа от `0` до `n-1` (опять последний элемент не включается!). В нём ровно `n` элементов. Ещё можно использовать `range` в двумя аргументами: указать начало и конец интервала: `range(3,9)`. Можно даже превратить `range` в настоящий список с помощью команды `list`:

In [ ]:

```
list(range(5))
```

In [ ]:

```
list(range(3,9))
```

In [ ]:

```

```