



*Федоров, Д. Ю. Программирование на языке высокого уровня Python : учеб. пособие для прикладного бакалавриата / Д. Ю. Федоров. — 2-е изд., перераб. и доп. — М. : Издательство Юрайт, 2019. — 161 с. — (Серия : Бакалавр. Прикладной курс).*

*Читать или заказать печатную версию можно на сайте издательства:*  
<https://biblio-online.ru/book/programmirovanie-na-yazyke-vysokogo-urovnya-python-437489>

<https://urait.ru/catalog/437489>

## ПРЕДИСЛОВИЕ

*«... руководители, не имеющие представления об ЭВМ и программировании, уйдут в небытие, профессиональные программисты станут системными аналитиками и системными программистами, а программировать сумеет каждый, что я и называю второй грамотностью».*

(1981 год, академик А.П. Ершов)

*«Техника сама по себе не поведет нас в нужном направлении, насколько я могу судить, ни в образовании, ни в социальной жизни. Я скорее сторонник революционных воззрений, чем реформист. Но революцию я предвижу в идеях, а не в технике».*

(1980 год, профессор Сеймур Пейперт)



В основу предлагаемого учебного пособия положен цикл видео-уроков и занятий, проведенных автором для студентов-экономистов СПбГЭУ, учеников лицея № 95 и слушателей курсов Epic Skills.

Цель пособия – рассказать об основах программирования для слушателей с минимальным знанием информатики. За 10-12 занятий данный курс позволяет научиться проектировать и разрабатывать приложения, используя базовые возможности языка программирования Python.

Язык программирования Python входит в пятерку по популярности в мире, поэтому найти по нему литературу не составит труда. На желающих стать программистами обрушится гора справочников и «лучших рекомендаций» по разработке приложений любого уровня сложности, но среди всех этих книг новичку бывает сложно разобраться, а первое знакомство с толстыми справочниками по внутреннему устройству Python может навсегда отпугнуть от занятия программированием.

На взгляд автора, не следует сваливать на головы учащихся сразу всю справочную информацию и множество правил, существующих в языках программирования – «не следует множить сущее без необходимости». Некоторые темы в пособии специально пришлось упростить, чтобы в вводном курсе не вдаваться в излишние детали, но в век Интернета поиск справочной информации не должен составить труда.

Автор благодарит всех, кто принял участие в разработке данного курса<sup>1</sup>.

Дмитрий Федоров, 5 марта 2016, г. Санкт-Петербург  
<http://dfedorov.spb.ru>

---

<sup>1</sup> Выражаю отдельную благодарность Максиму Петренко за поиск и исправление опечаток.

## ГЛАВА 1. ОСНОВЫ ОСНОВ

*«Информатика не более наука о компьютерах, чем астрономия – наука о телескопах».*

(Эдсгер Дейкстра)

*«Есть два типа языков программирования — те, которые все ругают, и те на которых никто не пишет».*

(Б. Страуструп, разработчик языка программирования C++)

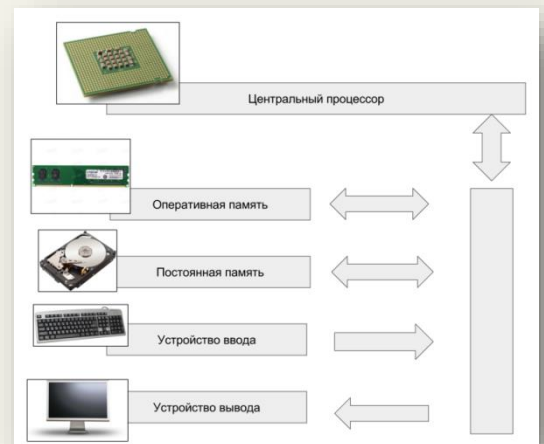
Для того чтобы научиться плавать необходимо войти в воду и начать пробовать грести руками, помогая себе ногами, затем побороть страх, оторваться от дна и поплыть. Есть в этом сходство с написанием программ. Можно прочесть толстый учебник, сдать зачет/экзамен в вузе, но при этом не научиться написанию даже простых программ.

Сколько времени тратить на обучение? Музыканты говорят, что для достижения мастерства владения инструментом необходимо репетировать по четыре часа в день.

Инструментом программиста является компьютер, поэтому кратко рассмотрим его устройство. Все вычисления в компьютере производятся центральным процессором. Файлы с программами хранятся в постоянной памяти (на жестком диске), а в момент выполнения загружаются во временную (оперативную) память. Ввод информации в компьютер осуществляется с помощью клавиатуры (устройства ввода), а вывод – с помощью монитора (устройства вывода).

Компьютер способен работать только с двумя видами сигналов: 1 или 0 (машинным кодом). Писать программы вида 10101010010101010 для человека сложно, мышление его устроено иначе, поэтому появились программы-трансляторы с языка программирования, понятного человеку, на машинный язык, понятный компьютеру.

Языки программирования, которые приближены к машинному уровню, называют языками низкого уровня (например, язык ассемблера). Другой вид языков – языки высокого уровня (например, Python, Java, C#), еще больше приближенные к мышлению человека.



вместе с ней – появилось множество программистов, для которых язык С стал родным. Написание программ на этом языке требует хорошей квалификации от программиста, т.к. незамеченная ошибка способна привести к серьезным последствиям в работе программы. До сих пор язык С лидирует в качестве языка для системного программирования.

Следующий этап (80-ые годы) характеризуется появлением объектно-ориентированного программирования (ООП), которое должно было упростить создание крупных промышленных программ. Появляется ученый – Б. Страуструп, которому недостаточно было возможностей языка С, поэтому он расширяет этот язык путем добавления ООП. Новый язык получил название С++.

В 90-ые годы появляются персональные компьютеры и сеть Интернет, потому требуются новые технологии и языки программирования. В этот момент набирает популярность язык Java, который позволяет в кратчайшие сроки начать писать крупные приложения без опасений что-либо серьезно испортить в системе. Язык Java создавался с оглядкой на С++ и с перспективой развития сети Интернет. Данный язык характеризуется переносимостью своих программ, т.е. написав Java-программу на персональном компьютере, можно запустить ее на кофемашине, если там присутствует виртуальная машина Java.

Примерно в одно время с Java появляется Python. Разработчик языка – математик Гвидо ван Россум занимался долгое время разработкой языка ABC, предназначенного для обучения программированию. В одном из интервью он так ответил на вопрос о типе программистов, для которых Python был бы интересен: *«Я представлял себе профессиональных программистов в UNIX или UNIX-подобной среде. Руководства для ранних версий Python возвещали что-то вроде «Python закрывает разрыв между Си и программированием оболочки», потому что именно это интересовало меня и моих ближайших коллег. Мне и в голову не приходило, что Python может стать хорошим языком для встраивания в приложения, пока меня не стали спрашивать об этом. То, что он оказался полезен для обучения началам программирования в школе или колледже, – счастливая случайность, обусловленная многими характеристиками ABC, которые я сохранил: ABC был специально предназначен для обучения программированию непрограммистов»*. К Python мы еще вернемся, а пока продолжим наш исторический экскурс.

С ростом сети Интернет потребовалось создавать динамические сайты – появился серверный язык программирования PHP, который на сегодняшний день является лидером при разработке веб-сайтов.

В 2000-ые годы наблюдается тенденция объединения технологий вокруг крупных корпораций. В это время получает развитие язык С# на платформе .NET.

## ГЛАВА 2. ЗНАКОМСТВО С ЯЗЫКОМ ПРОГРАММИРОВАНИЯ PYTHON

Чтобы читатель не подумал, что Python – игрушечный язык программирования, на котором можно только обучать основам программирования и дальше о нем благополучно забыть, кратко перечислю области, где он активно применяется:



1. Системное программирование.
2. Разработка программ с графическим интерфейсом.
3. Разработка динамических веб-сайтов.
4. Интеграция компонентов.
5. Разработка программ для работы с базами данных.
6. Быстрое создание прототипов.
7. Разработка программ для научных вычислений.
8. Разработка игр.

Что нам потребуется для выполнения программ на языке Python? Прежде, чем ответить на этот вопрос, рассмотрим, как запускаются программы на компьютере. Выполнение программ осуществляется операционной системой (Windows, Linux и пр.). В задачи операционной системы входит распределение ресурсов (оперативной памяти и пр.) для программы, запрет или разрешение на доступ к устройствам ввода/вывода и т.д.

Для запуска программ на языке Python необходима программа-интерпретатор (виртуальная машина) Python. Данная программа скрывает от Python-программиста все особенности



операционной системы, поэтому, написав

В математических выражениях в качестве операндов можно использовать целые числа<sup>5</sup> (1, 4, -5) или вещественные (в программировании их еще называют числами с плавающей точкой<sup>6</sup>): 4.111, -9.3. Математические операторы, доступные над числами в Python<sup>7</sup>:

Оператор	Описание
+	Сложение
-	Вычитание
/	Деление (в результате вещественное число)
//	Деление с округлением вниз
**	Возведение в степень
%	Остаток от деления

```
>>> 5 / 3
1.6666666666666667
>>> 5 // 3
1
>>> 5 % 3
2
>>> 5 ** 67
67762635780344027125465800054371356964111328125
>>>
```

Если один из операндов является вещественным числом, то в результате получится вещественное число.

В качестве упражнения найдите значение выражения  $2 + 56 * 5.0 - 45.5 + 5^5$ .

При вычислении математических выражений Python придерживается приоритета операций:

```
>>> -2**4
-16
>>> -(2**4)
-16
>>> (-2)**4
16
>>>
```

В случае сомнений в порядке вычислений будет не лишним обозначить приоритет в виде круглых скобок.

Выражаясь в терминах программирования, только что мы познакомились с числовым типом данных<sup>8</sup> (целым типом `int` и вещественным типом `float`), т.е. множеством числовых значений и множеством математических операций, которые можно выполнять над данными значениями. Язык Python предоставляет большой выбор встроенных типов данных.

---

<sup>5</sup> А также комплексные числа, логические значения: `True`, `False`

<sup>6</sup> Для знатоков языка C: числа с плавающей точкой в языке Python представлены обычными числами с плавающей точкой двойной точности (64 бита). Как правило, это представление соответствует стандарту IEEE 754, который позволяет обеспечивать представление примерно 17 значимых разрядов, с экспонентой в диапазоне от -308 до 308. Это полностью соответствует типу *double* в языке C.

<sup>7</sup> Интересно, что в Python выражение  $(b * (a // b) + a \% b)$  эквивалентно  $a$ .

<sup>8</sup> Забегая вперед скажем, что в объектно-ориентированном программировании типы данных называются классами.



В момент выполнения присваивания `cel = 26` в памяти компьютера создается объект, расположенный по некоторому адресу<sup>9</sup> (условно обозначим его как `id1`), имеющий значение 26 целочисленного типа `int`. Затем создается переменная с именем `cel`, которой присваивается адрес объекта `id1`. Переменные в Python содержат адреса объектов или можно сказать, что переменные ссылаются на объекты. Постоянно сохраняя в голове эту модель, для упрощения будем говорить, что переменная содержит значение.

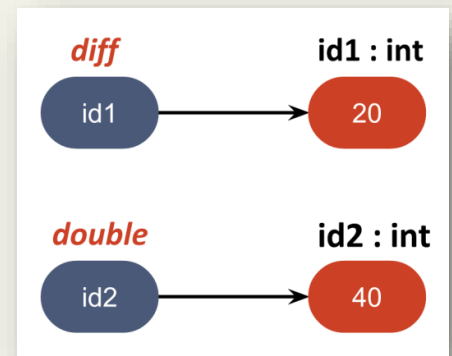
Вычисление следующего выражения в итоге приведет к присваиванию переменной `cel` значения 72, т.е. сначала вычисляется правая часть, затем результат присваивается левой части.

```
>>> cel = 26 + 46
>>> cel
72
>>>
```

Рассмотрим чуть более сложный пример. Вместо переменной `diff` подставится целочисленное значение 20:

```
>>> diff = 20
>>> double = 2 * diff
>>> double
40
>>>
```

По окончании вычислений память для Python будет иметь следующий вид:

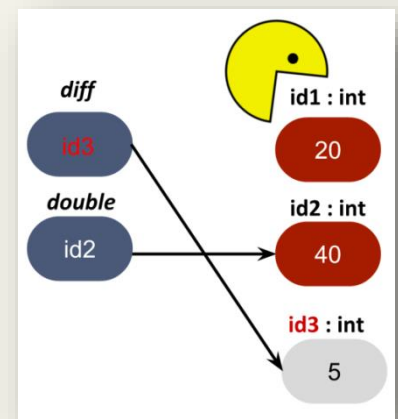


Продолжим вычисления. Присвоим переменной `diff` значение 5 и посмотрим содержимое переменных `double` и `diff`.

```
>>> diff = 5
>>> double
40
>>> diff
5
>>>
```

В момент присваивания переменной `diff` значения 5 в памяти создается объект по адресу `id3`, содержащий целочисленное значение 5. После этого изменится содержимое переменной `diff`, вместо адреса `id1` туда запишется адрес `id3`. Также Python увидит, что на объект по адресу `id1` больше никто не ссылается и поэтому удалит его из памяти (произведет автоматическую сборку мусора).

Внимательный читатель заметил, что Python не изменяет существующие числовые объекты, а создает



<sup>9</sup> **Информация для опытных программистов.** Функция `id` возвращает идентификатор объекта, переданного в качестве аргумента функции. В реализации CPython возвращаемое число является адресом объекта в памяти.

```
>>> abs(-9) + abs(5.6)
14.6
>>>
```

Результат вызова функции можно присвоить переменной, использовать его в качестве операндов математических выражений, т.е. составлять более сложные выражения.

Рассмотрим несколько популярных математических функций языка Python.

`pow(x, y)` возвращает значение  $x$  в степени  $y$ . Эквивалентно записи  $x^{**}y$ .

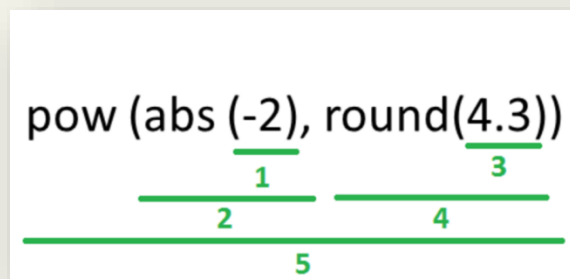
```
>>> pow(4, 5)
1024
>>>
```

`round(number)` возвращает число с плавающей точкой, округленное до 0 цифр после запятой (по умолчанию). Может быть вызвана с двумя аргументами:

`round(number[, ndigits])`, где `ndigits` – число знаков после запятой.

```
>>> round(4.56666)
5
>>> round(4.56666, 3)
4.567
>>>
```

Помимо составления сложных математических выражений Python позволяет передавать результаты вызова функций в качестве аргументов других функций без использования дополнительных переменных:



На рисунке представлен пример вызова и порядок вычисления выражений. В этом примере на месте числовых объектов `(-2, 4.3)` могут находиться вызовы функций или их комбинации, поэтому они тоже вычисляются.

Очень часто при написании программ требуется преобразовать объекты разных типов. Т.к. пока мы познакомились только с числовыми объектами, поэтому рассмотрим функции для их преобразования.

`int` возвращает целочисленный объект, построенный из числа или строки<sup>11</sup>, или 0, если аргументы не переданы.

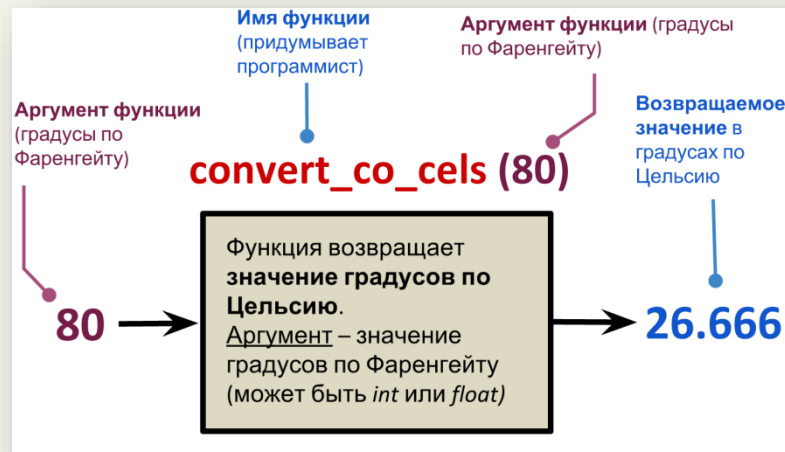
`float` возвращает число с плавающей точкой, построенное из числа или строки.

Рассмотрим примеры:

---

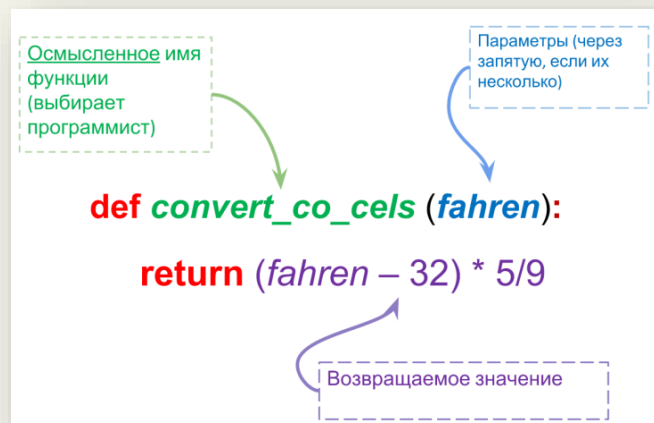
<sup>11</sup> Об этом типе данных в следующей главе





В первую очередь, необходимо придумать имя функции, к примеру, назовем функцию `convert_co_cels`. Постарайтесь, чтобы имя было осмысленным (`lena123` – плохой пример для имени функции) и отражало смысл функции, вспомните о правилах наименования переменных. Помимо этого, не желательно, чтобы имя вашей функции совпадало с именами встроенных функций Python (встроенные функции в IDLE подсвечиваются фиолетовым цветом).

Представим, что функция с именем `convert_co_cels` создана, тогда ее вызов для значения 80 будет иметь вид: `convert_co_cels(80)`.



Перейдем непосредственно к созданию функции. Ключевое слово `def` для Python означает, что дальше идет описание функции. После `def` указывается имя функции `convert_co_cels`, затем в скобках указывается параметр, которому будет присваиваться значение при вызове функции. Параметры функции – обычные переменные, которыми функция пользуется для внутренних вычислений. Переменные, объявленные внутри функции, называются *локальными* и не видны вне функции. После символа «:» начинается тело функции. В интерактивном режиме Python самостоятельно поставит отступ<sup>12</sup> от края экрана, тем самым обозначив, где начинается тело функции. Выражение, стоящее после ключевого слова `return` будет возвращаться в качестве результата вызова функции.

В интерактивном режиме создание функции имеет следующий вид (для завершения ввода функции необходимо два раза нажать клавишу <Enter>, дождавшись приглашения для ввода команд):

<sup>12</sup> Отступы играют важную роль в Python, отделяя тело функции, цикла и пр.

Создадим файл `myprog.py`, содержащий следующий код (тело функции должно отделяться четырьмя пробелами)<sup>13</sup>:

```
def f(x):  
    x = 2 * x  
    return x
```

Запустим программу с помощью F5. Увидим, что в интерактивном режиме программа выполнялась, но ничего не вывела на экран. Правильно, ведь мы не вызвали функцию!

```
===== RESTART: C:/Python35-32/myprog.py =====  
>>>
```

После запуска программы в интерактивном режиме вызовем функцию `f` с различными аргументами:

```
>>> f(4)  
8  
>>> f(56)  
112  
>>>
```

Все работает! Теперь вызовем функцию `f` в файле, но не забываем про `print`. Обновленная версия файла `myprog.py` будет иметь вид:

```
def f(x):  
    x = 2 * x  
    return x  
  
print(f(4))    # комментарии игнорируются Python  
print(f(56))
```

Запустим программу с помощью F5 и увидим, что в интерактивном режиме отобразился результат!

```
===== RESTART: C:/Python35-32/myprog.py =====  
8  
112  
>>>
```

Теперь поговорим об области видимости переменных. Ранее мы сказали, что переменная является локальной (видна только внутри функции), если значение ей присваивается внутри функций, в ином случае – переменная глобальная, т.е. видна (к ней можно обратиться) во всей программе, в том числе и внутри функции.

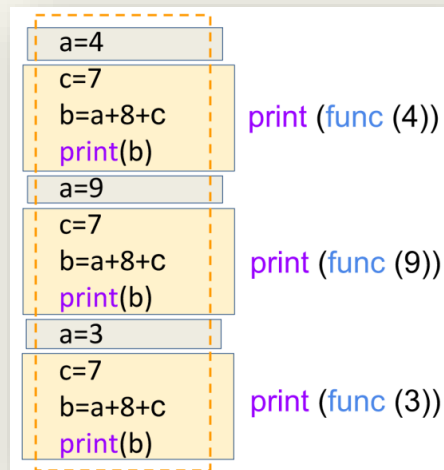
---

<sup>13</sup> Разработчики на языке Python очень трепетно относятся к оформлению исходного кода, поэтому следуют ряду правил, собранных в документе под названием PEP8.

Часто функции используются для сокращения кода программы, например, объявив функцию вида:

```
def func(x):  
    c = 7  
    return x + 8 + c
```

Следующий код может быть заменен на три вызова функции с различными аргументами:



В файле не забываем вызывать функцию `print`.

Бывают случаи, когда наша функция ничего не принимает на вход и ничего не возвращает<sup>16</sup> (не используется ключевое слово `return`). Пример подобной функции:

```
def print_hello():  
    print('Привет')  
    print('Hello')  
    print('Hi')
```

Видим, что внутри функции происходит вызов `print`, поэтому в момент вызова функции `print_hello` еще раз вызывать `print` не требуется. Следующий пример демонстрирует, что множество вызовов `print` можно заменить тремя вызовами функции `print_hello`:

---

<sup>16</sup> На самом деле, если не указать `return`, то Python вернет объект `None`:

```
>>> def f(x):  
    print(x**2 + 1)  
  
>>> f(2)  
5  
>>> 3*f(2)+1 # f(2) вернет объект None типа NoneType  
5  
Traceback (most recent call last):  
  File "<pyshell#14>", line 1, in <module>  
    3*f(2)+1  
TypeError: unsupported operand type(s) for *: 'int' and 'NoneType'  
>>>
```

```
def summa(x, y):  
    return x + y
```

```
def func(f, a, b):  
    return f(a, b)
```

```
v = func(summa, 10, 3) # передаем summa в качестве аргумента
```

Этот пример демонстрирует, как из функции `func` можно вызвать функцию `summa`.

Поиме этого, в момент вызова функции можно присваивать значения конкретным параметрам (использовать ключевые аргументы):

```
def func(a, b=5, c=10):  
    print('a равно', a, ', b равно', b, ', a c равно', c)
```

```
func(3, 7)           # a=3, b=7, c=10  
func(25, c=24)       # a=25, b=5, c=24  
func(c=50, a=100)    # a=100, b=5, c=50
```

Ошибкой будет являться вызов функции, при котором не задан аргумент **a**, т.к. для него не указано значение по умолчанию.

#### Упражнение 2.4

Напишите функцию в отдельном файле, вычисляющую среднее арифметическое трех чисел. Задайте значения по умолчанию, в момент вызова используйте ключевые аргументы.

Здесь начинаются удивительные вещи! Помните, мы говорили, что операции зависят от типа данных? Над объектами определенного типа можно производить только определенные операции: числа – складывать, умножать и т.д. Так вот, для строк символ + будет объединять строки, а для чисел – складывать их. А, что если сложить число и строку?

```
>>> 'Mapc' + 5
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    'Mapc' + 5
TypeError: Can't convert 'int' object to str implicitly
>>>
```

Python запутался, т.к. не понял, что мы от него хотим: сложить числа или объединить строки. К примеру, мы хотим объединить строки. Для этого с помощью функции `str` преобразуем число 5 в строку '5' и выполним объединение:

```
>>> 'Mapc' + str(5)
'Mapc5'
>>>
```

Можно ли выполнить обратное преобразование типов? Можно!

```
>>> int("-5")
-5
>>>
```

Попросим Python повторить нашу строку заданное число раз:

```
>>> "СПАМ" * 10
'СПАМСПАМСПАМСПАМСПАМСПАМСПАМСПАМСПАМ'
>>>
```

Операция умножения для строк приобрела другой смысл.

Строки можно присваивать переменным и дальше работать с переменными:

```
>>> s = "Я изучаю программирование"
>>> s
'Я изучаю программирование'
>>> s*4
'Я изучаю программированиеЯ изучаю программированиеЯ изучаю
программированиеЯ изучаю программирование'
>>> s + " на языке Python"
'Я изучаю программирование на языке Python'
>>>
```

Если хотим поместить разные виды кавычек в строку, то сделать это можно несколькими способами:

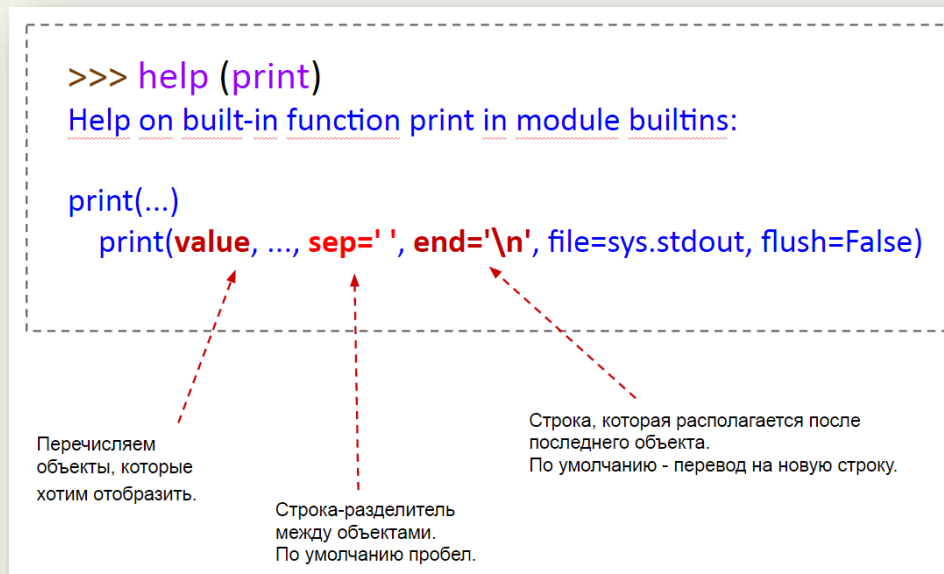
```
>>> "Hello's"
"Hello's"
>>> 'Hello\'s'
"Hello's"
>>>
```

Рассмотрим еще несколько примеров:

```
>>> print(1, 3, 5)
1 3 5
>>> print(1, '2', 'снова строка', 56)
1 2 снова строка 56
>>>
```

Убедились, что `print` позволяет выводить объекты разных типов.

На самом деле, у этой функции есть несколько «скрытых» аргументов, которые задаются по умолчанию в момент вызова:



И снова несколько примеров:

```
>>> print(1, 6, 7, 8, 9)
1 6 7 8 9
>>> print(1, 6, 7, 8, 9, sep=':')
1:6:7:8:9
>>>
```

По умолчанию `print` использует в качестве разделителя пробел, мы можем изменить разделитель, изменив для этого принудительно значение параметра `sep=':'`.

Отсюда вывод: полезно читать документацию.

Добавим в наши программы интерактивности, т.е. будем просить у пользователя ввести значение с клавиатуры. В Python для этого есть специальная функция `input`:

```
>>> s = input()
Земляне, мы прилетели с миром!
>>> s
'Земляне, мы прилетели с миром!'
>>> type(s)
<class 'str'>
>>>
```



Каждый символ строки имеет свой порядковый номер (*индекс*). Нумерация символов начинается с нуля. Теперь мы можем обратиться к заданному символу строки следующим образом:

```
>>> s = 'Я люблю писать программы!'
>>> s[0]
'Я'
>>> s[-1]
'!'
>>>
```

В квадратных скобках указывается индекс символа. Нулевой индекс – первая буква строки. А  $-1$  индекс? Можно догадаться, что последний. Если увидите отрицательный индекс, то определить его положительный аналог можно как длина строки + отрицательный индекс. Например, для  $-1$  это будет: `len(s) - 1`, т.е. 24.

```
>>> len(s) - 1
24
>>> s[24]
'!'
>>>
```

Какая ситуация с изменением строк в Python?

```
>>> s = 'Я люблю писать программы!'
>>> s[0] = 'J'
Traceback (most recent call last):
  File "<pyshell#41>", line 1, in <module>
    s[0] = 'J'
TypeError: 'str' object does not support item assignment
>>>
```

Попытка изменить нулевой символ в строке **s** привела к ошибке. Дело в том, что в Python строки, как и числа, являются неизменяемыми.

Работа со строковыми объектами для Python не отличается от работы с числовыми объектами:



```
>>> s[-1:]      # снова отрицательный индекс
'e'
>>>
```

Надеюсь, что срезы станут вашими верными помощниками при работе со строками.

Теперь вернемся к вопросу, как изменить первый символ в строке? Со срезами это «элементарно, Ватсон»!

```
>>> s = 'Я люблю писать программы!'
>>> 'J' + s[1:]
'J люблю писать программы!'
>>>
```

### Упражнение 3.2

Напишите программу, определяющую сумму и произведение трех чисел (типа `int`, `float`), введенных с клавиатуры.

Пример работы программы:

```
Введите первое число: 1
Введите второе число: 4
Введите третье число: 7
Сумма введенных чисел: 12
Произведение введенных чисел: 28
```

Например, высказывание: «6 — четное число». Истинно или ложно? Очевидно, что истинно. А высказывание: «6 больше 19»? Хм. Высказывание ложно. Никакого подвоха в этом нет.

Является ли высказыванием фраза: «у него голубые глаза»? Хочется спросить, у кого? Однозначности в этой фразе нет, поэтому она не является высказыванием.

Далее высказывания можно комбинировать. Высказывания «Петров — врач», «Петров — шахматист» можно объединять с помощью связок И, ИЛИ.

«Петров — врач И шахматист». Это высказывание истинно, если ОБА высказывания «Петров — врач» И «Петров — шахматист» являются истинными.

«Петров — врач ИЛИ шахматист». Это высказывание истинно, если истинным является ОДНО ИЗ высказываний «Петров — врач» ИЛИ «Петров — шахматист».

Как это используется в Python? Рассмотрим пример комбинаций из высказываний:

```
>>> 2 > 4
False
>>> 45 > 3
True
>>> 2 > 4 and 45 > 3 # комбинация False and True вернет False
False
>>> 2 > 4 or 45 > 3 # комбинация False or True вернет True
True
>>>
```

Все, что мы сказали про комбинацию логических высказываний, можно объединить и представить в виде таблицы<sup>22</sup>, где 0 — False, а 1 — True.

X Y	and	or
0 0	0	0
0 1	0	1
1 0	0	1
1 1	1	1

Для Python истинным или ложным может быть не только логическое высказывание, но и объект. Так, что же такое истина в Python?

---

<sup>22</sup> Ее называют таблицей истинности.

```
>>>
```

Результатом применения логического оператора `not` (НЕ) произойдет отрицание операнда, т.е. если операнд истинный, то `not` вернет – ложь, если ложный, то – истину.

Логический оператор `and` (И) вернет `True` (истину) или `False` (ложь)<sup>24</sup>, если его операндами являются логические высказывания.

```
>>> 2 > 4 and 45 > 3 # комбинация False and True вернет False
False
>>>
```

Если операндами оператора `and` являются объекты, то в результате Python вернет объект:

```
>>> '' and 2 # False and True
''
>>>
```

Для вычисления оператора `and` Python вычисляет операнды слева направо и возвращает первый объект, имеющий ложное значение.

Посмотрим на столбец `and` таблицы истинности. Какая закономерность? Если среди операндов (X, Y) есть ложный, то получим ложное значение, но вместо ложного значения для операндов-объектов Python возвращает первый ложный операнд, встретившийся в выражении, и дальше вычисления НЕ производит. Это называется вычислением по короткой схеме.

```
>>> 0 and 3 # вернет первый ложный объект-операнд
0
>>> 5 and 4 # вернет крайний правый объект-операнд
4
>>>
```

Если Python не удастся найти ложный объект-операнд, то он возвращает крайний правый операнд.

Логический оператор `or` действует похожим образом, но для объектов-операндов Python возвращает первый объект, имеющий истинное значение. Python прекратит дальнейшие вычисления, как только будет найден первый объект, имеющий истинное значение.

```
>>> 2 or 3 # вернет первый истинный объект-операнд
2
>>> None or 5 # вернет второй объект-операнд, т.к. первый всегда ложный
5
>>> None or 0 # вернет оставшийся объект-операнд
0
```

X Y	and	or
0 0	0	0
0 1	0	1
1 0	0	1
1 1	1	1

Таким образом, конечный результат становится известен еще до вычисления остальной части выражения.

---

<sup>24</sup> Исходя из таблицы истинности

Строки в Python тоже можно сравнивать по аналогии с числами. Начнем издалека. Символы, как и все остальное, представлено в компьютере в виде чисел. Есть специальная таблица, которая ставит в соответствие каждому символу некоторое число.

Определить, какое число соответствует символу можно с помощью функции `ord`:

```
>>> ord('L')
76
>>> ord('Ф')
1060
>>> ord('A')
65
>>>
```

Теперь сравнение символов сводится к сравнению чисел, которые им соответствуют:

```
>>> 'A' > 'L'
False
>>>
```

Для сравнения строк Python их сравнивает посимвольно:

```
>>> 'Aa' > 'Ll'
False
>>>
```

Следующий полезный оператор, с которым мы познакомимся – `in`. Он проверяет наличие подстроки в строке:

```
>>> 'a' in 'abc'
True
>>> 'A' in 'abc'      # большой буквы А нет
False
>>> "" in 'abc'       # пустая строка есть в любой строке
True
>>> '' in ''
True
>>>
```

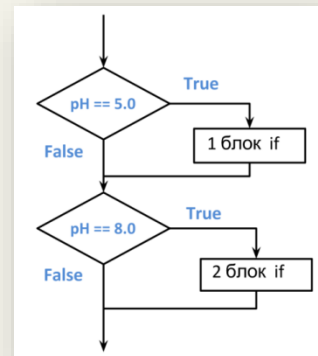
Освоив логические операции, перейдем к их использованию.

Можно производить несколько проверок подряд, и они выполняются по очереди:

```
>>> pH = 5.0
>>> if pH == 5.0:
    print(pH, "Кофе")

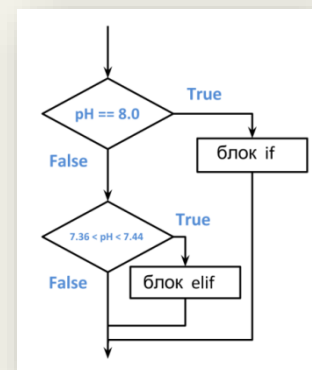
5.0 Кофе
>>> if pH == 8.0:
    print(pH, "Вода")

>>>
```



Часто встречаются задачи, где выполнять все проверки не имеет смысла. Следующую программу наберите и выполните в отдельном файле (не забывайте про отступы для блока выражений `if`, их должно быть четыре):

```
pH = 3.0
if pH == 8.0:
    print(pH, "Вода")
elif 7.36 < pH < 7.44:
    print(pH, "Кровь")
```



В этой программе используется ключевое слово `elif` (сокращение от `else if`), которое проверяет условие `7.36 < pH < 7.44`, если `pH == 8.0` оказалось ложным. Графически это представлено на блок-схеме алгоритма, расположенной справа от программы.

Условное выражение может включать множество проверок. Общий синтаксис у него следующий:

```
if << условие >>:
    << блок выражений >>
elif << условие >>:
    << блок выражений >>
else:
    << блок выражений >>
```

Блок выражений, относящийся к `else`, выполняется, когда все вышестоящие условия вернули `False`.



В `"""` тройные двойные кавычки в теле функции помещается информация, которую выводит на экран функция `help`. Теперь вы можете добавлять описание к собственным функциям.

### Упражнение 5.1

Напишите программу для определения индекса массы тела (BMI).

### Упражнение 5.2

Напишите собственную программу, определяющую максимальное из двух введенных чисел. Реализовать в виде вызова собственной функции, возвращающей большее из двух переданных ей чисел.

### Упражнение 5.3

Напишите программу, проверяющую целое число на четность. Реализовать в виде вызова собственной функции.

### Упражнение 5.4

Напишите программу, вычисляющую значение функции (на вход подается вещественное число):

$$f = \begin{cases} x^2 & \text{при } -2,4 \leq x \leq 5,7, \\ 4 & \text{в противном случае.} \end{cases}$$

### Упражнение 5.5

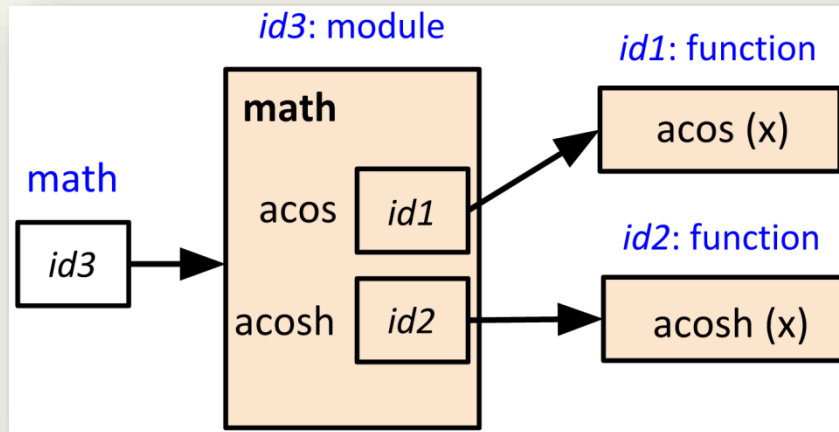
Напишите программу, которая по коду города и длительности переговоров вычисляет их стоимость и результат выводит на экран: Екатеринбург-код 343, 15 руб/мин; Омск-код 381, 18 руб/мин; Воронеж-код 473, 13 руб/мин; Ярославль-код 485, 11 руб/мин.

В момент импортирования модуля `math` создается переменная с именем `math`:

```
>>> type(math)
<class 'module'>
>>>
```

Функция `type` показала, что тип данных переменной `math` – модуль.

Переменная `math` содержит ссылку (адрес) модульного объекта. В этом объекте содержатся ссылки на функции (функциональные объекты):



В момент вызова функции `sqrt` Python находит переменную `math` (модуль должен быть предварительно импортирован), просматривает модульный объект, находит функцию `sqrt` внутри этого модуля и затем выполняет ее.

В Python можно импортировать отдельную функцию из модуля:

```
>>> from math import sqrt
>>> sqrt(9)
3.0
>>>
```

Таким образом, Python не будет создавать переменную `math`, а загрузит в память только функцию `sqrt`. Теперь вызов функции можно производить, не обращаясь к имени модуля `math`. Здесь надо быть крайне внимательным. Приведу пример, почему:

```
>>> def sqrt(x):
    return x * x

>>> sqrt(5)
25
>>> from math import sqrt
>>> sqrt(9)
3.0
>>>
```

## ГЛАВА 7. СОЗДАНИЕ СОБСТВЕННЫХ МОДУЛЕЙ

Теперь попробуем создать собственный модуль.

Создайте файл с именем `mm.py` (для модулей обязательно указывается расширение `.py`) и содержащий код (содержимое нашего модуля):

```
def f():  
    return 4
```

Теперь нужно сказать Python, где искать наш модуль. Выясним через обращение к переменной `path` модуля `sys`, где Python по умолчанию хранит собственные модули (у вас список каталогов может отличаться):

```
>>> import sys  
>>> sys.path  
['', 'C:\\Python35-32\\Lib\\idlelib', 'C:\\Python35-32\\python35.zip', 'C:\\Python35-32\\DLLs', 'C:\\Python35-32\\lib', 'C:\\Python35-32', 'C:\\Python35-32\\lib\\site-packages']  
>>>
```

Далее поместим наш модуль в один из перечисленных каталогов, например, в `'C:\\Python35-32'`.

Если мы все правильно сделали, то импортируем наш модуль, указав только его имя (без расширения):

```
>>> import mm  
>>> mm.f()  
4  
>>>
```

Ура-ура! Теперь мы через точку можем вызывать функцию, которая находится в модуле `mm`.

Продолжим изучение модулей в Python. Создадим еще один модуль (по аналогии с предыдущим), укажем для него другое имя – `mtest.py`:

```
print('test')
```

Новый модуль будет содержать вызов функции `print`. Импортируем его несколько раз подряд:

```
>>> import mtest  
test  
>>> import mtest  
>>>
```

Что мы видим? Во-первых, импортирование модуля выполняет содержащиеся в нем команды. Во-вторых, повторное импортирование не приводит к выполнению модуля, т.е. он повторно не импортируется. Объясняется это тем, что импортирование модулей в память – ресурсоемкий процесс, поэтому лишний раз Python его не производит. Но как быть, если

Если мы запускаем модуль, то содержимое переменной `__name__` будет равно строке `__main__`, а в случае импортирования – переменная `__name__` будет содержать имя модуля.

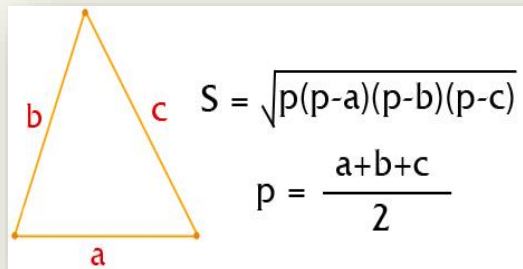
Рассмотрим, как это использовать на практике. Создадим модуль с именем `prog3.py` и содержанием:

```
def func(x):  
    return x**2+7  
  
if __name__ == "__main__":  
    x = int(input("Введите значение: "))  
    print(func(x))
```

Теперь Python поймет, когда мы хотим выполнить модуль, а когда – импортировать. Если модуль выполнить (Run → Run Module), то выполнится весь файл, т.к. сработает условие `if`. При импортировании модуля (`import prog3`) условие не выполнится и Python загрузит в память только функцию `func`. Попробуйте проделать это самостоятельно.

### Упражнение 7.1

Найдите площадь треугольника с помощью формулы Герона. Стороны задаются с клавиатуры. Реализовать вычисление площади в виде функции, на вход которой подаются три числа, на выходе – площадь. Функция находится в отдельном модуле, где происходит разделение между запуском и импортированием. Описание математических функций можно найти в документации<sup>25</sup>



### Упражнение 7.2

Вывести число Пи с точностью до сотых.

### Упражнение 7.3

Создайте в отдельном модуле функцию для вычисления выражения:

$$\sqrt{1 - \sin^2 x}$$

<sup>25</sup> <https://docs.python.org/3/library/math.html>

```
def func_m(v1, v2, v3):
    """Вычисляет среднее арифметическое трех чисел.

    >>> func_m(20, 30, 70)
    60.0

    >>> func_m(1, 5, 8)
    6.667

    """
    return round((v1+v2+v3)/3, 3)

import doctest
# автоматически проверяет тесты в документации
doctest.testmod()
```

В результате выполнения программы получим:

```
>>>
===== RESTART: C:/Python35-32/mypr.py =====
*****
File "C:/Python35-32/mypr.py", line 4, in __main__.func_m
Failed example:
    func_m(20, 30, 70)
Expected:
    60.0
Got:
    40.0
*****
File "C:/Python35-32/mypr.py", line 7, in __main__.func_m
Failed example:
    func_m(1, 5, 8)
Expected:
    6.667
Got:
    4.667
*****
1 items had failures:
  2 of   2 in __main__.func_m
***Test Failed*** 2 failures.
>>>
```

Теперь вы умеете создавать собственные тесты!

### Упражнение 7.5

Напишите функцию, вычисляющую значение:

$$x^4 + 4^x$$

Автоматизируйте процесс тестирования функции с помощью модуля doctest.

## ГЛАВА 8. СТРОКОВЫЕ МЕТОДЫ В PYTHON

Вызовем функцию `type` и передадим ей на вход целочисленный аргумент:

```
>>> type(0)
<class 'int'>
>>>
```

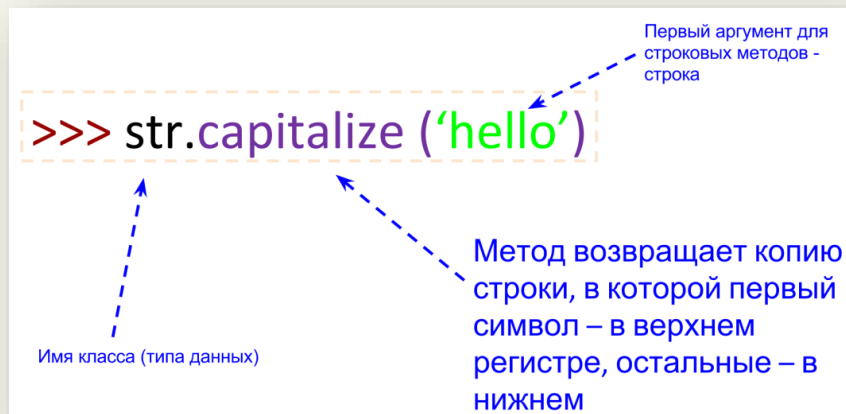
Функция сообщила нам, что объект `0` относится к классу `'int'`, т.е. **тип данных является классом** (тип данных и класс – синонимы).

Мы еще не рассматривали ООП, поэтому класс будем представлять, как некий аналог модуля, т.е. набор функций и переменных, содержащихся внутри класса. **Функции, которые находятся внутри класса, называются методами**. Их главное отличие от вызова функций из модуля заключается в том, что в качестве первого аргумента метод принимает, например, строковый объект, если это метод строкового класса.

Рассмотрим пример вызова строкового метода:

```
>>> str.capitalize('hello')
'Hello'
>>>
```

По аналогии с вызовом функции из модуля указываем имя класса – `str`, затем через точку пишем имя строкового метода `capitalize`, который принимает один строковый аргумент:



Метод – это обычная функция, расположенная внутри класса. Вызовем еще один метод:

```
>>> str.center('hello', 20)
'      hello      '
>>>
```

Этот метод принимает два аргумента – строку и число:



Вынесенный из метода первый строковый аргумент может быть выражением, возвращающим строку:

```
>>> ('ТТА' + 'G'*3).count('T')
2
>>>
```

Не сложно догадаться, что делает метод count.  
Python содержит интересный метод format<sup>27</sup>:

```
>>> '{0} и {1}'.format('труд', 'май')
'труд и май'
>>>
```

Вместо {0} и {1} подставляются аргументы метода format.  
Поменяем их местами:

```
>>> '{1} и {0}'.format('труд', 'май')
'май и труд'
>>>
```

Формат вывода метода format может варьироваться:

```
>>> n = 10
>>> '{:b}'.format(n)      # вывод в двоичной системе счисления
'1010'
>>> '{:c}'.format(n)      # вывод в формате Unicode
'\n'
>>> '{:d}'.format(n)      # по снованию 10
'10'
>>> '{:x}'.format(n)      # по основанию 16
'a'
>>>
```

В Python есть полезные строковые методы, которые возвращают (True) истину или (False) ложь:

```
>>> 'spec'.startswith('a')
False
>>>
```

Метод startswith проверяет, начинается ли строка с символа, переданного в качестве аргумента методу.

При работе с текстами полезно использовать строковый метод strip:

```
>>> s = '          \n ssssss  \n'
>>> s.strip()
'ssssss'
>>>
```

---

<sup>27</sup> <https://docs.python.org/3.1/library/string.html#format-examples>

## Для справки. Специальные строковые методы

Объединим две строки:

```
>>> 'TT' + 'rr'
'TTrr'
>>>
```

На самом деле, в этот момент Python вызывает специальный строковый метод `__add__` и передает ему в качестве первого аргумента строку `'rr'`:

```
>>> 'TT'.__add__('rr')
'TTrr'
>>>
```

Напомню, что этот вызов затем преобразуется Python в полную форму (результат будет аналогичный):

```
>>> str.__add__("TT", 'rr')
'TTrr'
>>>
```

Забегая вперед скажу, что за каждой из операций над типами данных стоит свой специальный метод.

### Упражнение 8.1

```
s = "У лукоморья 123 дуб зеленый 456"
```

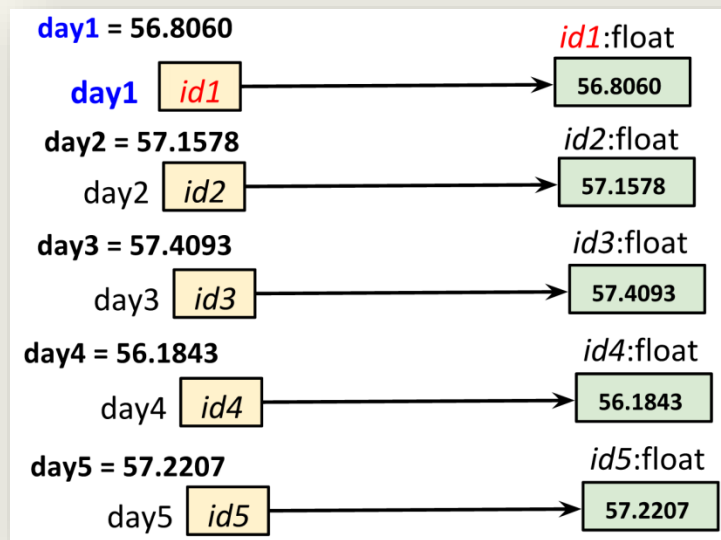
1. Определить, встречается ли в строке буква 'я'. Вывести на экран ее позицию (индекс) в строке.
2. Определить, сколько раз в строке встречается буква 'у'.
3. Определить, состоит ли строка только из букв, ЕСЛИ нет, ТО вывести строку в верхнем регистре.
4. Определить длину строки. ЕСЛИ длина строки превышает 4 символа, ТО вывести строку в нижнем регистре.
5. Заменить в строке первый символ на 'О'. Результат вывести на экран

### Упражнение 8.2

Написать в отдельном модуле функцию, которая на вход принимает два аргумента: строку (`s`) и целочисленное значение (`n`).

ЕСЛИ длина строки `s` превышает `n` символов, ТО функция возвращает строку `s` в верхнем регистре, ИНАЧЕ возвращается исходная строка `s`.

Схематично:



А, если обработать необходимо курсы валют за последние два года...?

Тут нам на помощь приходят **списки**. Их можно рассматривать как аналог массива в других языках программирования, за исключением важной особенности – списки в качестве своих элементов могут содержать любые объекты. Но обо всем по порядку.

Список (`list`) в Python является объектом<sup>30</sup>, поэтому может быть присвоен переменной (переменная, как и в предыдущих случаях, хранит адрес объекта класса список).

Представим список для нашей задачи с курсом валют:

```
>>> e = [56.8060, 57.1578, 57.4093, 56.1843, 57.2207]
>>> e
[56.806, 57.1578, 57.4093, 56.1843, 57.2207]
>>>
```

Список позволяет хранить разнородные данные, обращаться к которым можно через имя списка (в данном случае переменную `e`).

<sup>30</sup> В Python все является объектами

Обращение по несуществующему индексу вызовет ошибку:

```
>>> e[100]
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    e[100]
IndexError: list index out of range
>>>
```

До настоящего момента мы рассматривали типы данных (классы), которые нельзя было изменить. Вспомните, как Python ругался при попытке изменить строку.

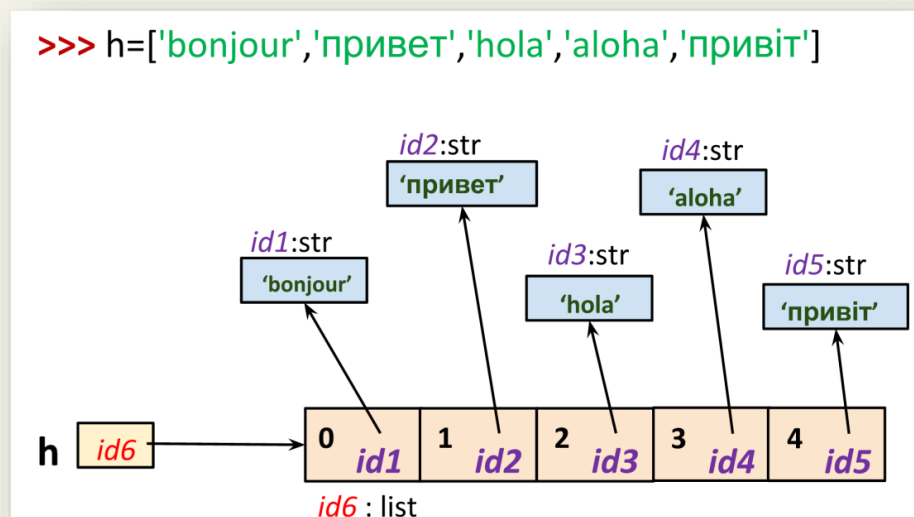
Списки можно изменить. Проведем эксперимент:

```
>>> h = ['Hi', 27, -8.1, [1, 2]]
>>> h[1] = 'hello'
>>> h
['Hi', 'hello', -8.1, [1, 2]]
>>> h[1]
'hello'
>>>
```

В примере мы создали список и изменили элемент, находящийся в позиции 1. Видим, что список изменился. Рассмотрим еще один пример и покажем, что происходит в памяти:

```
>>> h = ['bonjour', 'привет', 'hola', 'aloha', 'привіт']
>>>
```

В памяти:



Производим изменения списка:

```
>>> h[1] = 'hello'
>>> h
['bonjour', 'hello', 'hola', 'aloha', 'привіт']
>>> h[1]
'hello'
>>>
```

### Упражнение 9.1

```
L = [3, 6, 7, 4, -5, 4, 3, -1]
```

1. Определите сумму элементов списка L. ЕСЛИ сумма превышает значение 2, ТО вывести на экран число элементов списка.
2. Определить разность между минимальным и максимальным элементами списка. ЕСЛИ абсолютное значение разности больше 10, ТО вывести на экран отсортированный по возрастанию список, ИНАЧЕ вывести на экран фразу «Разность меньше 10».

Операция + для списков служит для их объединения (вспомните строки):

```
>>> original = ['H', 'B']
>>> final = original + ['T']
>>> final
['H', 'B', 'T']
```

Операция повторения (снова аналогия со строками):

```
>>> final = final * 5
>>> final
['H', 'B', 'T', 'H', 'B', 'T', 'H', 'B', 'T', 'H', 'B', 'T', 'H', 'B', 'T']
```

Инструкция del позволяет удалять из списка элементы по индексу:

```
>>> del final[0]
>>> final
['B', 'T', 'H', 'B', 'T', 'H', 'B', 'T', 'H', 'B', 'T', 'H', 'B', 'T']
```

Рассмотрим интересный пример, но для начала напишите функцию, объединяющую два списка.

Получится следующее:

```
>>> def f(x, y):
        return x + y

>>> f([1, 2, 3], [4, 5, 6])
[1, 2, 3, 4, 5, 6]
>>>
```

Теперь передадим в качестве аргументов две строки:

```
>>> f("123", "456")
'123456'
>>>
```

Передадим два числа:

```
>>> f(1, 2)
3
```

Вернемся к инструкции `del` и удалим с помощью среза подсписок:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]          # удаление подсписка
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
>>>
```

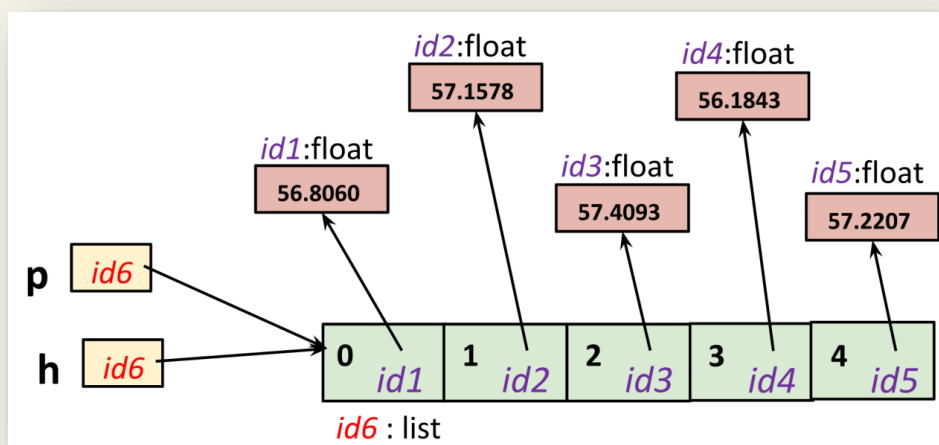
### 9.3. Псевдонимы и копирование списков

Рассмотрим важную особенность списков. Выполним следующий код:

```
>>> h
['bonjour', 7, 'hola', -1.0, 'привіт']
>>> p = h      # содержат указатель на один и тот же список
>>> p
['bonjour', 7, 'hola', -1.0, 'привіт']
>>> p[0] = 1    # модифицируем одну из переменных
>>> h           # изменилась другая переменная!
[1, 7, 'hola', -1.0, 'привіт']
>>> p
[1, 7, 'hola', -1.0, 'привіт']
>>>
```

В Python две переменные называются *псевдонимами*<sup>32</sup>, когда они содержат одинаковые адреса памяти.

На схеме видно, что переменные `p` и `h` указывают на один и тот же список:



<sup>32</sup> Псевдонимы – альтернативные имена чего-либо



Второй вид копирования – *глубокое копирование*. При глубоком копировании создается новый объект и рекурсивно создаются копии всех объектов, содержащихся в оригинале<sup>34</sup>.

```
>>> import copy
>>> a = [4, 3, [2, 1]]
>>> b = copy.deepcopy(a)
>>> b[2][0] = -100
>>> a
[4, 3, [2, 1]] # список a не изменился
>>>
```

С одной стороны список предоставляет возможность модификации, с другой – появляется опасность незаметно изменить список за счет создания псевдонимов или при поверхностном копировании.

### Упражнение 9.3

```
L = [3, 'hello', 7, 4, 'привет', 4, 3, -1]
```

Исследуйте несколько примеров использования срезов (выполняются аналогично строкам).

```
>>> L[:3]
>>> L[:]
>>> L[:2]
>>> L[:-1]
>>> L[-1]
>>> L[-1:]
```

### 9.4. Методы списка

Вспомните, что мы говорили о строковых методах. Для списков ситуация будет аналогичная. Далее приведены наиболее популярные методы списка<sup>35</sup>:

```
>>> colors = ['red', 'orange', 'green']
>>> colors.extend(['black', 'blue']) # расширяет список списком
>>> colors
['red', 'orange', 'green', 'black', 'blue']
>>> colors.append('purple') # добавляет элемент в список
>>> colors
['red', 'orange', 'green', 'black', 'blue', 'purple']
>>> colors.insert(2, 'yellow') # добавляет элемент в указанную позицию
>>> colors
['red', 'orange', 'yellow', 'green', 'black', 'blue', 'purple']
>>> colors.remove('black') # удаляет элемент из списка
>>> colors
['red', 'orange', 'yellow', 'green', 'blue', 'purple']
>>> colors.count('red') # считает количество повторений аргумента метода
1
>>> colors.index('green') # возвращает позицию в списке аргумента метода
3
```

---

<sup>34</sup> <https://docs.python.org/3/library/copy.html>

<sup>35</sup> <https://docs.python.org/3/tutorial/datastructures.html#more-on-lists>

Метод `join` принимает на вход список, который необходимо преобразовать в строку, а в качестве строкового объекта указывается соединитель элементов списка.

Аналогично можно преобразовать число к списку (через строку) и затем изменить полученный список:

```
>>> n = 73485384753846538465
>>> list(str(n)) # число преобразуем в строку, затем строку в список
['7', '3', '4', '8', '5', '3', '8', '4', '7', '5', '3', '8', '4',
'6', '5', '3', '8', '4', '6', '5']
>>>
```

Если строка содержит разделитель, то ее можно преобразовать к списку с помощью строкового метода `split`, который по умолчанию в качестве разделителя использует пробел:

```
>>> s = 'd a dd dd gg rr tt yy rr ee'.split()
>>> s
['d', 'a', 'dd', 'dd', 'gg', 'rr', 'tt', 'yy', 'rr', 'ee']
>>>
```

Возьмем другой разделитель:

```
>>> s = 'd:a:dd:dd:gg:rr:tt:yy:rr:ee'.split(":")
>>> s
['d', 'a', 'dd', 'dd', 'gg', 'rr', 'tt', 'yy', 'rr', 'ee']
>>>
```

#### Упражнение 9.4

```
L = [3, 'hello', 7, 4, 'привет', 4, 3, -1]
```

Определите наличие строки «привет» в списке. ЕСЛИ такая строка в списке присутствует, ТО удалить ее из списка, ИНАЧЕ добавить строку в список.

Подсчитать, сколько раз в списке встречается число 4, ЕСЛИ больше одного раза, ТО очистить список.

### 9.6. Вложенные списки

Мы уже упоминали, что в качестве элементов списка могут быть объекты любого типа, например, списки:

```
>>> lst = [['A', 1], ['B', 2], ['C', 3]]
>>> lst
[['A', 1], ['B', 2], ['C', 3]]
>>> lst[0]
['A', 1]
>>>
```

Подобные структуры используются для хранения матриц.

## ГЛАВА 10. ИНСТРУКЦИИ ЦИКЛА В PYTHON

Язык Python позволяет быстро создавать прототипы<sup>36</sup> реальных программ благодаря тому, что в него заложены конструкции для решения типовых задач, с которыми часто приходится сталкиваться программисту.

Вспомните, как мы решали задачу подсчета суммы элементов списка через вызов функции `sum([1, 4, 5, 6, 7.0, 3, 2.0])` – всего лишь один вызов функции!

В этой главе мы рассмотрим еще несколько подобных приемов, которые значительно упрощают жизнь разработчика на языке Python.

### 10.1. Инструкция цикла *for*

Например, у нас имеется список `num` и мы хотим красиво вывести на экран каждый из его элементов:

```
>>> num = [0.8, 7.0, 6.8, -6]
>>> num
[0.8, 7.0, 6.8, -6]
>>> print(num[0], '- number')
0.8 - number
>>> print(num[1], '- number')
7.0 - number
>>> # AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA!
```

Если в списке будет пятьсот элементов?! Для подобных случаев в Python существуют циклы. Циклы являются движущей силой в программировании. Их понимание позволит писать настоящие живые и полезные программы!

Перепишем этот пример с использованием цикла `for`:

```
>>> num = [0.8, 7.0, 6.8, -6]
>>> for i in num:
    print(i, '- number')

0.8 - number
7.0 - number
6.8 - number
-6 - number
>>>
```

Цикл `for` позволяет перебрать все элементы указанного списка. Цикл сработает ровно столько раз, сколько элементов находится в списке. Имя переменной, в которую на каждом шаге будет помещаться элемент списка, выбирает программист. В нашем примере это переменная с именем `i`.

На первом шаге переменной `i` будет присвоен первый элемент списка `num`, равный 0.8. Затем программа переходит в тело цикла `for`, отделенное отступами (четыре пробела или одна табуляция). В теле цикла содержится вызов функции `print`, которой передается переменная `i`.

На следующем шаге переменной `i` присвоится второй элемент списка, равный 7.0. Произойдет вызов функции `print` для отображения содержимого переменной `i` на экране и т.д. до тех пор, пока не закончатся элементы в списке!

---

<sup>36</sup> Быстрая, черновая реализация будущей программы.

```
7.0 - число 7.0  
>>>
```

Например, можем вывести на экран только заданное значение из списка, выполнив сравнение на каждом шаге цикла.

Похожим образом в цикле производится поиск необходимого символа в строке с помощью вызова строкового метода:

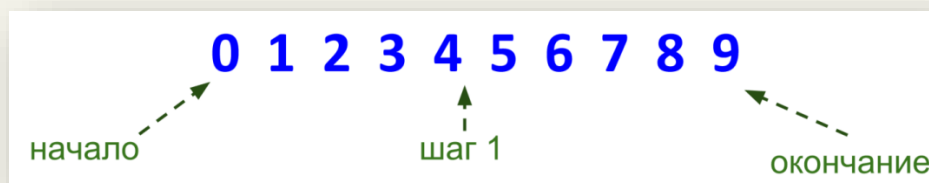
```
>>> country = "Russia"  
>>> for ch in country:  
    if ch.isupper():  
        print(ch)
```

```
R  
>>>
```

Напоминаю, что строковый метод `isupper` проверяет верхний регистр символа (С БОЛЬШОЙ ЛИ ОН БУКВЫ?), возвращает `True` или `False`. В цикле проверяется каждый символ строки. Если символ в верхнем регистре, то он выводится на экран.

## 10.2. Функция *range*

Достаточно часто при разработке программ необходимо получить последовательность (диапазон) целых чисел:



Для решения этой задачи в Python предусмотрена функция `range`, создающая последовательность (диапазон) чисел. В качестве аргументов функция принимает: начальное значение диапазона (по умолчанию 0), конечное значение (не включительно) и шаг (по умолчанию 1). Если вызвать функцию, то результата мы не увидим:

```
>>> range(0, 10, 1)  
range(0, 10)  
>>> range(10)  
range(0, 10)  
>>>
```

В Python есть более красивое решение данной задачи:

```
>>> sum(list(range(1, 101))) # sum(range(1, 101))
5050
>>>
```

Это решение требует небольших пояснений. Диапазоны можно использовать при создании списков:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(2, 10, 2))
[2, 4, 6, 8]
>>>
```

Вызов функции `sum` для списка в качестве аргумента приводит к подсчету суммы всех элементов списка – это как раз то, что нам нужно!

### Упражнение 10.1

Найдите все значения функции  $y(x) = x^2 + 3$  на интервале от 10 до 30 с шагом 2.

### Упражнение 10.2

```
L = [-8, 8, 6.0, 5, 'строка', -3.1]
```

Определить сумму чисел, входящих в список `L`. *Подсказка:* для определения типа объекта можно воспользоваться сравнением вида `type(-8) == int`.

Диапазон, создаваемый функцией `range`, часто используется для задания индексов. Например, если необходимо изменить существующий список, умножив каждый его элемент на 2:

```
lst = [4, 10, 5, -1.9]
print(lst)
for i in range(len(lst)):
    lst[i] = lst[i] * 2
print(lst)
```

В результате выполнения программы:

```
>>>
===== RESTART: C:/Python35-32/myprog.py =====
[4, 10, 5, -1.9]
[8, 20, 10, -3.8]
>>>
```

Необходимо пройти в цикле по всем элементам списка `lst`, для этого перебираются и изменяются последовательно элементы списка через указание их индекса. В качестве аргумента `range` задается длина списка. В этом случае создаваемый диапазон будет от 0 до `len(lst) - 1`. Python не включает крайний элемент диапазона, т.к. длина списка всегда на 1 больше, чем индекс последнего его элемента, т.к. индексация начинается с нуля.

### 10.3. Подходы к созданию списка

```
>>> a = [2, -2, 4, -4, 7, 5]
>>> b = [i**2 for i in a]
>>> b
[4, 4, 16, 16, 49, 25]
>>>
```

В примере мы выбираем последовательно значения из списка `a`, возводим в квадрат каждый из его элементов и сразу добавляем полученные значения в новый список.

По аналогии можно перебирать символы из строки и формировать из них список:

```
>>> c = [c*3 for c in 'list' if c != 'i']
>>> c
['lll', 'sss', 'ttt']
>>>
```

В Python есть интересная функция `map`, которая позволяет создавать новый список на основе существующего списка:

```
>>> def f(x):
        return x + 5

>>> list(map(f, [1, 3, 4]))
[6, 8, 9]
>>>
```

Функция `map` принимает в качестве аргументов имя функции и список (или строку). Каждый элемент списка (или строки) подается на вход функции, и результат работы функции добавляется как элемент нового списка. Получить результат вызова функции `map` можно через цикл `for` или функцию `list`. Функции, которые принимают на вход другие функции, называются *функциями высшего порядка*.

Пример вызова `map` для строки:

```
>>> def f(s):
        return s * 2

>>> list(map(f, "hello"))
['hh', 'ee', 'll', 'll', 'oo']
>>>
```

Рассмотрим, как получить список, состоящий из случайных целых чисел:

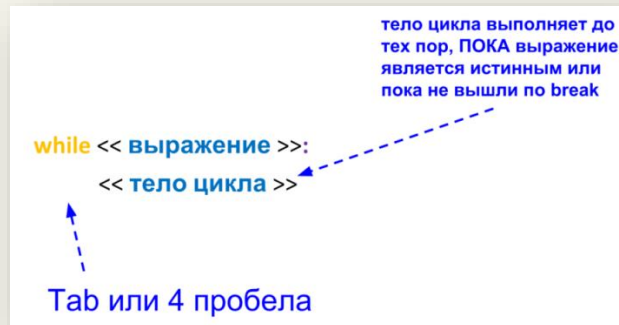
```
>>> from random import randint
>>> A = [randint(1, 9) for i in range(5)]
>>> A
[2, 1, 1, 7, 8]
>>>
```

В данном примере функция `range` выступает как счетчик числа повторений (цикл `for` сработает ровно 5 раз). Обратите внимание, что при формировании нового списка переменная `i` не используется. В результате пять раз будет произведен вызов функции

## 10.4. Инструкция цикла *while*

Как вы уже догадались, цикл `for` используется, если заранее известно, сколько повторений необходимо выполнить (указывается через аргумент функции `range` или пока не закончится список/строка).

Если заранее количество повторений цикла неизвестно, то применяется другая конструкция, которая называется циклом `while`:



Определим количество кроликов:

```
rabbits = 3  
while rabbits > 0:  
    print(rabbits)  
    rabbits = rabbits - 1
```

В результате выполнения программы:

```
>>>  
===== RESTART: C:\Python35-32\myprog.py =====  
3  
2  
1  
>>>
```

В примере цикл `while` выполняется до тех пор, ПОКА число кроликов в условии положительное. На каждом шаге цикла мы переменную `rabbits` уменьшаем на 1, чтобы не уйти в бесконечный цикл, когда условие всегда будет являться истинным.

Рассмотрим подробнее ход выполнения программы.

В начале работы программы переменная `rabbits` равна 3, затем попадаем в цикл `while`, т.к. условие `rabbits > 0` будет являться истинным (вернет значение `True`). В теле цикла вызывается функция `print`, которая отобразит на экране текущее значение переменной `rabbits`. Далее переменная уменьшится на 1 и снова произойдет проверка условия `while`, т.е. `2 > 0` (вернет `True`). Попадаем в цикл и действия повторяются до тех пор, пока не дойдем до условия `0 > 0`. В этом случае вернется логическое значение `False` и цикл `while` не сработает.

Результат выполнения:

```
>>>
===== RESTART: C:\Python35-32\myprog.py =====
сумма чисел: 9
>>>
```

В примере демонстрируется использование инструкции `continue`. Выполнение данной инструкции приводит к переходу к следующему шагу цикла, т.е. все команды, которые находятся после `continue`, будут проигнорированы.

#### Упражнение 10.5

Дано число, введенное с клавиатуры. Определите сумму квадратов нечетных цифр в числе.

#### Упражнение 10.6

Найдите сумму чисел, вводимых с клавиатуры. Количество вводимых чисел заранее неизвестно. Окончание ввода, например, слово «Стоп».

#### Упражнение 10.7

Задана строка из стихотворения: «Мой дядя самых честных правил, Когда не в шутку занемог, Он уважать себя заставил И лучше выдумать не мог»

Удалите из строки все слова, начинающиеся на букву «м». Результат вывести на экран в виде строки.

Подсказка: вспомните про модификацию списков.

#### Упражнение 10.8

Дан произвольный текст. Найдите номер первого самого длинного слова в нем.

#### Упражнение 10.9

Дан произвольный текст. Напечатайте все имеющиеся в нем цифры, определите их количество, сумму и найти максимальное.



Сначала пример с одним циклом `for`:

```
lst = [[1, 2, 3],  
       [4, 5, 6]]
```

```
for i in lst:  
    print(i)
```

Результат выполнения программы:

```
>>>  
===== RESTART: C:\Python35-32\myprog.py =====  
[1, 2, 3]  
[4, 5, 6]  
>>>
```

В примере с помощью цикла `for` перебираются все элементы списка, которые также являются списками.

Если мы хотим добраться до элементов вложенных списков, то придется использовать вложенный цикл `for`:

```
lst = [[1, 2, 3],  
       [4, 5, 6]]
```

```
for i in lst:      # цикл по элементам внешнего списка  
    print()  
    for j in i:    # цикл по элементам элементов внешнего списка  
        print(j, end="")
```

Результат выполнения программы:

```
>>>  
===== RESTART: C:\Python35-32\myprog.py =====  
  
123  
456  
>>>
```

### Упражнение 10.10

Создайте матрицу (список из вложенных списков) размера  $N \times M$  (фиксируются в программе), заполненную случайными целыми числами.

### Упражнение 10.11

Создайте матрицу (список из вложенных списков) размера  $N \times N$  (фиксируются в программе), заполненную случайными целыми числами.

### Упражнение 10.12

Дана матрица (см. упражнение 10.10). Вывести номер строки, содержащей максимальное число одинаковых элементов.

## ГЛАВА 11. ДОПОЛНИТЕЛЬНЫЕ ТИПЫ ДАННЫХ В PYTHON

### 11.1. Множества

Математическое образование разработчика языка Python наложило свой отпечаток на типы данных (классы), которые присутствуют в языке.

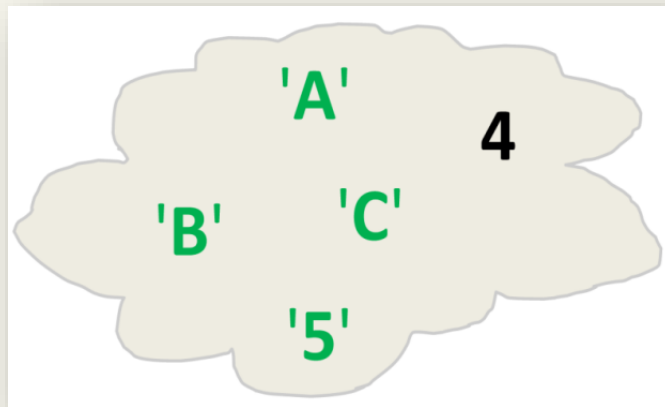
Рассмотрим множество (set) в Python – неупорядоченную коллекцию неизменяемых, уникальных элементов.

Создадим множество:

```
>>> v = {'A', 'C', 4, '5', 'B'}
>>> v
{'C', 'B', '5', 4, 'A'}
>>>
```

Заметим, что полученное множество отобразилось не в том порядке, в каком мы его создавали, т.к. множество – это неупорядоченная коллекция.

Представим множество схематично:



Множества в Python обладают интересными свойствами:

```
>>> v = {'A', 'C', 4, '5', 'B', 4}
>>> v
{'C', 'B', '5', 4, 'A'}
>>>
```

Видим, что повторяющиеся элементы, которые мы добавили при создании множества, были удалены (элементы множества уникальны).

Рассмотрим способы создания множеств:

```
>>> set([3, 6, 3, 5])
{3, 5, 6}
>>>
```

Множества можно создавать на основе списков. Обратите внимание, что в момент создания множества из списка будут удалены повторяющиеся элементы. Это отличный способ очистить список от повторов:

```
>>> list(set([3, 6, 3, 5]))
```

## 11.2. Кортежи

Следующий тип данных (класс), который также уходит своими корнями в математику – кортеж (tuple). Кортеж условно можно назвать неизменяемым «списком», т.к. к нему применимы многие списковые функции, кроме изменения. Кортежи используются, когда мы хотим быть уверены, что элементы структуры данных не будут изменены в процессе работы программы. Вспомните проблему с псевдонимами у списков.

Некоторые операции над кортежами<sup>38</sup>:

```
>>> ()      # создание пустого кортежа
()
>>> (4)      # это не кортеж, а целочисленный объект!
4
>>> (4,)     # а вот это – кортеж, состоящий из одного элемента!
(4,)
>>> b = ('1', 2, '4') # создаем кортеж
>>> b
('1', 2, '4')
>>> len(b)   # определяем длину кортежа
3
>>> t = tuple(range(10)) # создание кортежа с помощью функции range()
>>> t + b     # слияние кортежей
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, '1', 2, '4')
>>> r = tuple([1, 5, 6, 7, 8, '1']) # кортеж из списка
>>> r
(1, 5, 6, 7, 8, '1')
>>>
```

С помощью кортежей можно присваивать значения одновременно двум переменным:

```
>>> (x, y) = (10, 5)
>>> x
10
>>> y
5
>>> x, y = 1, 3 # если убрать круглые скобки, то результат не изменится
>>> x
1
>>> y
3
>>>
```

Поменять местами содержимое двух переменных:

```
>>> x, y = y, x
>>> x
3
>>> y
1
>>>
```

---

<sup>38</sup> <https://docs.python.org/3/tutorial/datastructures.html#tuples-and-sequences>

В качестве индексов словаря используются неизменяемые строки, могли бы воспользоваться кортежами, т.к. они тоже неизменяемые:

```
>>> e = {}
>>> e
{}
>>> e[(4, '6')] = '1'
>>> e
{(4, '6'): '1'}
>>>
```

Результирующий словарь `eng2sp` отобразился в «перемешанном» виде, т.к. по аналогии с множествами, словари являются неупорядоченной коллекцией.

К словарям применим оператор `in`:

```
>>> eng2sp
{'three': 'tres', 'one': 'uno', 'two': 'dos'}
>>> 'one' in eng2sp    # поиск по множеству КЛЮЧЕЙ
True
>>>
```

Часто словари используются, если требуется найти частоту встречаемости элементов в последовательности (списке, строке, кортеже<sup>39</sup>).

Функция, которая возвращает словарь, содержащий статистику встречаемости элементов в последовательности:

```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] = d[c] + 1 # или d[c] += 1
    return d
```

Результат вызова функции `histogram` для списка, строки, кортежа соответственно:

```
>>> histogram([2, 5, 6, 5, 4, 4, 4, 4, 3, 2, 2, 2])
{2: 5, 3: 1, 4: 4, 5: 2, 6: 1}
>>> histogram("ywte3475eryt3478e477477474")
{'4': 6, '8': 1, 'e': 3, '3': 2, '7': 7, '5': 1, 'r': 1, 'y': 2, 'w': 1, 't': 2}
>>> histogram((5, 5, 5, 6, 5, 'r', 5))
{5: 5, 6: 1, 'r': 1}
>>>
```

---

<sup>39</sup> Вы, наверно, обратили внимание, что все эти типы данных имеют общие свойства, поэтому их относят к последовательностям.

## ГЛАВА 12. НЕСКОЛЬКО СЛОВ ОБ АЛГОРИТМАХ

В предыдущих главах мы рассмотрели основные типы данных (классы), которые Python предоставляет программисту для работы. Теперь несколько слов отдельно скажем об алгоритмах.

Алгоритм – конечный набор шагов, который требуется для выполнения задачи, например, алгоритм заваривания чая или алгоритм похода в магазин. Каждая функция в программе и каждая программа – это реализация определенного алгоритма, написанного на языке программирования.

К примеру, нам необходимо найти позицию наименьшего элемента в следующем наборе данных: 809, 834, 477, 478, 307, 122, 96, 102, 324, 476.

Первым делом выбираем подходящий для хранения тип данных. Очевидно, что это будет список:

```
>>> counts = [809, 834, 477, 478, 307, 122, 96, 102, 324, 476]
>>> counts.index(min(counts)) # решение задачи в одну строку!
6
>>>
```

Усложним задачу и попытаемся найти позицию двух наименьших элементов в не отсортированном списке.

Какие возможны алгоритмы решения?

1. Поиск, удаление, поиск. Поиск индекса минимального элемента в списке, удаление его, снова поиск минимального, возвращаем удаленный элемент в список.
2. Сортировка, поиск минимальных, определение индексов.
3. Перебор всего списка. Сравниваем каждый элемент по порядку, получаем два наименьших значения, обновляем значения, если найдены наименьшие.

Рассмотрим каждый из перечисленных алгоритмов.

**1. Поиск, удаление, поиск:** поиск индекса минимального элемента в списке, удаление его, снова поиск минимального, возвращаем удаленный элемент в список.  
Начнем:

	[809, 834, 477, 478, 307, 122, 96, 102, 324, 476]									
индекс:	0	1	2	3	4	5	6	7	8	9
<i>Первый минимальный : 96</i>										
<i>Индекс элемента 96 : 6</i>										

Удаляем из списка найденный минимальный элемент. При этом **индексы в обновленном списке смещаются**:

В отличие от человека, который может охватить взглядом сразу весь список и моментально сказать, какой из элементов является минимальным, компьютер не обладает подобным интеллектом. Машина просматривает элементы по одному, последовательно перебирая и сравнивая элементы.

На первом шаге просматриваем первые два элемента списка:

	<b>[809, 834, ... ]</b>	
<b>индекс:</b>	<b>0</b>	<b>1</b>

Сравниваем 809 и 834 и определяем наименьший из них, чтобы задать начальные значения `min1` и `min2`, где будут храниться индексы первого минимального и второго минимального элементов соответственно.

Затем перебираем элементы, начиная со 2-ого индекса до окончания списка:

	<b>809</b>	<b>834</b>
	<hr/>	
	<code>min1</code>	<code>min2</code>

Определили, что 809 – первый минимальный, а 834 – второй минимальный элемент из двух первых встретившихся значений списка.

Просматриваем следующий элемент списка (477):

	<b>809</b>	<b>834</b>	<b>[..., 477, ... ]</b>
	<hr/>		
	<code>min1</code>	<code>min2</code>	

Элемент 477 оказался меньше всех (условно назовем это «первым вариантом»):

<b>477</b>	<b>809</b>	<b>834</b>
	<hr/>	
	<code>min1</code>	<code>min2</code>

Поэтому обновляем содержимое переменных `min1` и `min2`, т.к. нашли новый наименьший элемент:

<b>477</b>	<b>809</b>
<hr/>	
<code>min1</code>	<code>min2</code>

Специально не останавливался на теории построения и оценки алгоритмов. Приведу книги, где об этом говорится хорошо и подробно.

1. Томас Х. Кормен. *Алгоритмы. Вводный курс.*
2. Томас Х. Кормен. *Алгоритмы. Построение и анализ.*
3. Стивен С. Скиена. *Алгоритмы. Руководство по разработке.*

#### Упражнение 12.1

Напишите функцию, которая возвращает разность между наибольшим и наименьшим значениями из списка целых случайных чисел.

#### Упражнение 12.2

Напишите программу, которая для целочисленного списка из 1000 случайных элементов определяет, сколько отрицательных элементов располагается между его максимальным и минимальным элементами.

В упражнениях 12.3-8 список состоит из случайных элементов, в списке не менее 1000 элементов.

#### Упражнение 12.3

Найти элемент, наиболее близкий к среднему значению всех элементов списка.

#### Упражнение 12.4

Дан список, состоящий из чисел. Найти сумму простых чисел в списке.

#### Упражнение 12.5

Дан список целых чисел. Определить, есть ли в нем хотя бы одна пара соседних нечетных чисел. В случае положительного ответа определить номера элементов первой из таких пар.

#### Упражнение 12.6

Дан список целых чисел. Определить количество четных элементов и количество элементов, оканчивающихся на цифру 5.

#### Упражнение 12.7

Задан список из целых чисел. Определить процентное содержание элементов, превышающих среднеарифметическое всех элементов списка.

#### Упражнение 12.8

Задан список из целых чисел. Определить количество участков списка, на котором элементы монотонно возрастают (каждое следующее число больше предыдущего).

#### Упражнение 12.9

Дан список из 20 элементов. Найти пять соседних элементов, сумма значений которых максимальна.

Перепишем наш пример с учетом возможностей Python:

```
try:
    x = int(input("Enter number: "))
    print(5/x)
except:
    print("Error dividing by zero")
```

Выполним программу:

```
>>>
===== RESTART: C:\Python35-32\test.py =====
Enter number: 0
Error dividing by zero
>>>
```

В блок `try` помещается код, в котором может произойти ошибка. В случае возникновения ошибки (исключения) управление передается в блок `except`. Запустим программу и увидим, что при возникновении ошибки перевода буквы в число, мы снова попадаем в блок `except`:

```
>>>
===== RESTART: C:\Python35-32\test.py =====
Enter number: t
Error dividing by zero
>>>
```

Дело в том, что `except` без указания типа перехватываемой ошибки (исключения) обрабатывает все виды ошибок. Как нам разделить ошибки деления на нуль и преобразования типов?

Перейдем в интерактивный режим Python и выполним несколько команд, приводящих к ошибкам (исключениям):

```
>>> 4/0
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    4/0
ZeroDivisionError: division by zero
>>> int("r")
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    int("r")
ValueError: invalid literal for int() with base 10: 'r'
>>>
```

При делении на нуль возникает ошибка `ZeroDivisionError`, а при преобразовании типов – `ValueError`.



Отмечу только, что информацию об исключении можно помещать в переменную (с помощью инструкции `as`) и выводить на экран с помощью функции `print`.

Перехват исключений используется при написании функций, например:

```
def list_find(lst, target):
    try:
        index = lst.index(target)
    except ValueError:
        ## ValueError: value is not in list
        index = -1
    return index

print(list_find([3, 5, 6, 7], -6))
```

Результат выполнения:

```
>>>
===== RESTART: C:\Python35-32\1.py =====
-1
>>>
```

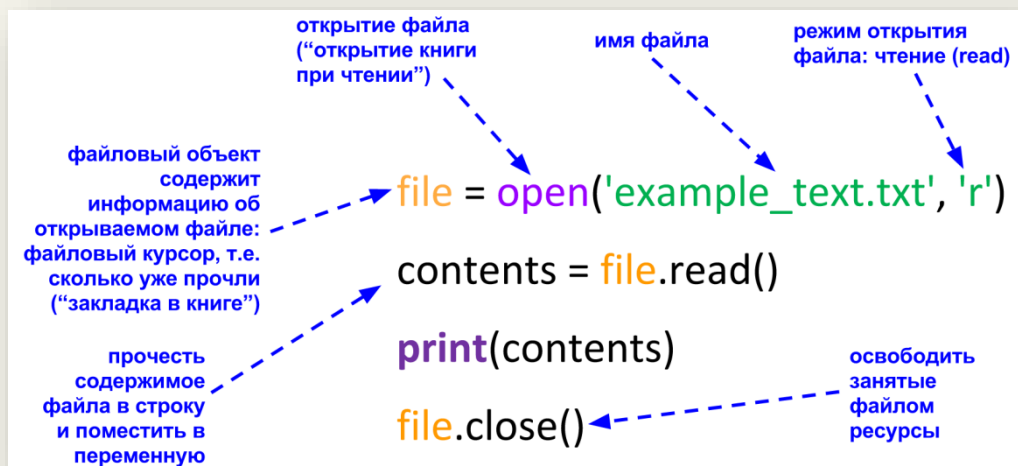
### Упражнение 13.1

Напишите программу, проверяющую четность числа, вводимого с клавиатуры. Выполните обработку возможных исключений.

### Упражнение 13.2

Напишите программу, которая будет генерировать матрицу из случайных целых чисел. Пользователь может указать число строк и столбцов, а также диапазон целых чисел. Произведите обработку ошибок ввода пользователя.

Небольшие комментарии к исходному тексту:



Рассмотренный подход по работе с файлами в Python перешел из языка C.

По умолчанию, если не указывать режим открытия, то используется открытие на «чтение». Файлы особенно подвержены ошибкам во время работы с ними. Диск может заполниться, пользователь может удалить используемый файл во время записи, файл могут переместить и т.д. Эти типы ошибок можно перехватить с помощью обработки исключений:

```
# Ошибка при открытии файла
try:
    f = open('example_text.txt') # открытие на чтение
except:
    print("Error opening file")
else: # выполняется в любом случае
    f.close()
    print('(Очистка: Закрытие файла)')
```

Запустим программу:

```
>>>
=== RESTART: C:\Python35-32\file_examples\file_reader.py ===
Error opening file
>>>
```

В дальнейшем для работы с файлами мы будем использовать *менеджер контекста* (инструкцию `with`<sup>41</sup>), который не требует ручного освобождения ресурсов.

Перепишем предыдущий пример с использованием менеджера контекста:

```
try:
    with open('example_text.txt', 'r') as file:
        contents = file.read()
        print(contents)
except:
    print("Error opening file")
```

Выполним программу:

---

<sup>41</sup> Менеджер контекста используется не только при работе с файлами.

Результат выполнения:

```
>>>
=== RESTART: C:\Python35-32\file_examples\file_reader.py ===
First line of text
Second line of text
Third line of text
>>>
```

Следующий пример демонстрирует работу с курсором:

```
with open('example_text.txt', 'r') as file:
    contents = file.read(10) # указываем кол-во символов для чтения
    # курсор перемещается на 11 символ
    rest = file.read()      # читаем с 11 символа
print("10:", contents)
print("остальное:", rest)
```

Результат работы программы:

```
>>>
=== RESTART: C:\Python35-32\file_examples\file_reader.py ===
10: First line
остальное:  of text
Second line of text
Third line of text
>>>
```

Если необходимо получить список, состоящий из строк, то можно воспользоваться методом `readlines`:

```
with open('example_text.txt', 'r') as file:
    lines = file.readlines()
print(lines)
```

Результат работы программы:

```
>>>
===== RESTART: C:\Python35-32\file_examples\file_reader.py =====
['First line of text\n', 'Second line of text\n', 'Third line of text']
>>>
```

Для демонстрации следующего примера создайте файл *plan.txt*, содержащий следующий текст:

```
Mercury
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune
```

Результат выполнения:

```
>>>
==== RESTART: C:\Python35-32\file_examples\file_reader.py ====
Mercury

7
Venus

5
Earth

5
Mars

4
Jupiter

7
Saturn

6
Uranus

6
Neptune
7
>>>
```

Следующий пример производит запись строки в файл. Если файла с указанным именем в рабочем каталоге нет, то он будет создан, если файл с таким именем существует, то он будет ПЕРЕЗАПИСАН:

```
with open("top.txt", 'w') as output_file:
    output_file.write("Hello!\n")
    # метод write возвращает число записанных символов
```

Для добавления строки в файл необходимо открыть файл в режиме «a» (сокр. от append):

```
with open("top.txt", 'a') as output_file:
    output_file.write("Hello!\n")
```

### Упражнение 14.1

Отсортированное по алфавиту содержимое файла *plan.txt* поместите в файл *sort\_plan.txt*.

## Упражнение 14.5

Определите три наиболее популярных вида спорта в стране, исходя из количества построенных спортивных объектов для них.

Файл с данными находится по адресу: <http://dfedorov.spb.ru/python3/sport.txt>

Номер	Наименование	Полный адрес	Виды спорта	Пропускная способность	Вместимость	Площадь (Га)
1	Государственное бюджетное учреждение города Москвы Спортивный комплекс Крылатское Москомспорта	Крылатская ул, 16, Москва, Россия, 121609	конькобежный спорт		7209	12,7275
2	Горнолыжный комплекс Узы-Тау	Октябрьский, Россия, 452613	сноуборд , горнолыжный спорт	310	0	36,57
3	Здание спортивно-тренировочного центра прикладных видов спорта	Кирова, 100, Медынь, Россия, 249950	волейбол , самбо	500	200	1,0826
4	Футбольное поле с искусственным покрытием	Девонская, 12 А, Октябрьский, Россия, 452600			0	1,2
5	Горнолыжный центр с инженерной защитой территории, хребет Аябга, урочище Роза Хутор	Красная поляна, Сочи, Россия, 354392	горнолыжный спорт , спортивный туризм	10450	0	21
6	Открытое Акционерное Общество Стадион СПАРТАК	Фрунзе, 15, Новосибирск, Россия, 630091	волейбол	350	13487	7,1024
7	Ледовый стадион «Сокол» Бюджетное образовательное учреждение Чувашской республики дополнительного образования детей «Специализированная детско-юношеская спортивная школа олимпийского резерва № 4 по хоккею с шайбой» Министерства по физической культуре, спорту и туризму Чувашской республики	Жени Крутовой, 1А, Новочебоксарск, Россия, 429951		120	1800	4,325

Поля (столбцы) файла:

- порядковый номер строки;
- наименование спортивного объекта;
- полный адрес спортивного объекта;
- виды спорта, для которых предназначен спортивный объект;
- пропускная способность объекта;
- вместимость объекта;
- площадь объекта (Га).

Разделитель между полями (столбцами) в файле: '\t'

Кодировка файла: 'cp1251'

PS: для сбора статистики можно воспользоваться словарем (dict), после чего провести его сортировку по значению.

Сортировка списка:

```
>>> sorted("This is a test string from Andrew".split())
['Andrew', 'This', 'a', 'from', 'is', 'string', 'test']
```

```
>>> str.lower("a")
'a'
>>> str.lower('Andrew')
'andrew'
```

Сортировка с предварительным применением к каждому элементу списка строкового метода lower. Метод указывается в качестве значения параметра key.

```
>>> sorted("This is a test string from Andrew".split(), key=str.lower)
['a', 'Andrew', 'from', 'is', 'string', 'test', 'This']
```

## ГЛАВА 15. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ В PYTHON

### 15.1. Основы объектно-ориентированного подхода

Ранее мы говорили о том, что Python является полностью объектно-ориентированным языком программирования, но подробно не рассматривали, что это означает. Вернемся к этому вопросу, начнем с примера.

Предположим, что существует набор строковых переменных для описания адреса проживания некоторого человека:

```
addr_name = 'Ivan Ivanov'    # имя человека
addr_line1 = '1122 Main Street'
addr_line2 = ''
addr_city = 'Panama City Beach'
addr_state = 'FL'
addr_zip = '32407'          # индекс
```

Напишем функцию, которая выводит на экран всю информацию о человеке:

```
def printAddress(name, line1, line2, city, state, zip):
    print(name)
    if len(line1) > 0:
        print(line1)
    if len(line2) > 0:
        print(line2)
    print(city + ", " + state + " " + zip)
```

**# Вызов функции, передача аргументов:**

```
printAddress(addr_name, addr_line1, addr_line2, addr_city, addr_state, addr_zip)
```

В результате работы программы:

```
>>>
===== RESTART: C:/Python35-32/addr.py =====
Ivan Ivanov
1122 Main Street
Panama City Beach, FL 32407
>>>
```

Предположим, что изменились начальные условия и у человека в адресе появился второй индекс. Почему бы и нет? Создадим новую переменную:

```
# добавим переменную, содержащую индекс
addr_zip2 = "678900"
```

Зададим поля объекта, адрес которого находится в переменной vacationHomeAddress:

```
vacationHomeAddress.name = "Ivan Ivanov"
vacationHomeAddress.line1 = "1122 Main Street"
vacationHomeAddress.line2 = ""
vacationHomeAddress.city = "Panama City Beach"
vacationHomeAddress.state = "FL"
vacationHomeAddress.zip = "32407"
```

Выведем на экран информацию о городе для основного и загородного адресов проживания (через указание имен объектов):

```
print("Основной адрес проживания " + homeAddress.city)
print("Адрес загородного дома " + vacationHomeAddress.city)
```

Изменим исходный текст функции printAddress() с учетом полученных знаний об объектах:

```
def printAddress(address): # передаем в функцию объект
    print(address.name)    # выводим на экран поле объекта
    if len(address.line1) > 0:
        print(address.line1)
    if len(address.line2) > 0:
        print(address.line2)
    print(address.city + ", " + address.state + " " + address.zip)
```

Если объекты homeAddress и vacationHomeAddress ранее были созданы, то можем вывести информацию о них, передав в качестве аргумента функции printAddress:

```
printAddress(homeAddress)
printAddress(vacationHomeAddress)
```

В результате выполнения программы получим:

```
>>>
===== RESTART: C:/Python35-32/addr2.py =====
Ivan Ivanov
701 N. C Street
Carver Science Building
Indianola, IA 50125
Ivan Ivanov
1122 Main Street
Panama City Beach, FL 32407
>>>
```

Возможности классов и объектов не ограничиваются лишь объединением переменных под одним именем, т.е. хранением состояния объекта. Классы также позволяют задавать функции внутри себя (методы) для работы с полями класса, т.е. влиять на поведение объекта.

Создадим класс Dog:

```
class Dog:
    age = 0      # возраст собаки
    name = ""    # имя собаки
    weight = 0   # вес собаки
    # Первым аргументом любого метода всегда является self, т.е. сам объект
    def bark(self): # функция внутри класса называется методом
```

Рассмотрим пример присвоения имени собаки через вызов конструктора класса:

```
class Dog:
    name = ""
    # Конструктор
    # Вызывается на момент создания объекта этого типа
    def __init__(self, newName):
        self.name = newName

# Создаем собаку и устанавливаем ее имя:
myDog = Dog("Spot")

# Вывести имя собаки, убедиться, что оно было установлено
print(myDog.name)

# Следующая команда выдаст ошибку, потому что
# конструктору не было передано имя
# herDog = Dog()
```

Результат работы программы:

```
>>>
===== RESTART: C:/Python35-32/dog2.py =====
Spot
>>>
```

Теперь имя собаки присваивается в момент ее создания. В конструкторе указали `self.name`, т.к. в момент вызова конструктора вместо `self` подставится конкретный объект, т.е. `myDog`.

В предыдущем примере для обращения к имени собаки мы выводили на экран поле `myDog.name`, т.е., переводя на язык реального мира, мы залезали во внутренности объекта и доставали оттуда информацию. Звучит жутковато, поэтому обеспечим «гуманные» методы для работы с именем объекта-собаки (`setName` и `getName`):

```
class Dog:
    name = ""
    # Конструктор вызывается в момент создания объекта этого класса
    def __init__(self, newName):
        self.name = newName
    # Можем в любой момент вызвать метод и изменить имя собаки
    def setName(self, newName):
        self.name = newName
    # Можем в любой момент вызвать метод и узнать имя собаки
    def getName(self):
        return self.name # возвращаем текущее имя объекта

# Создаем собаку с начальным именем:
myDog = Dog("Spot")

# Выводим имя собаки:
print(myDog.getName())

# Установим новое имя собаки:
myDog.setName("Sharik")

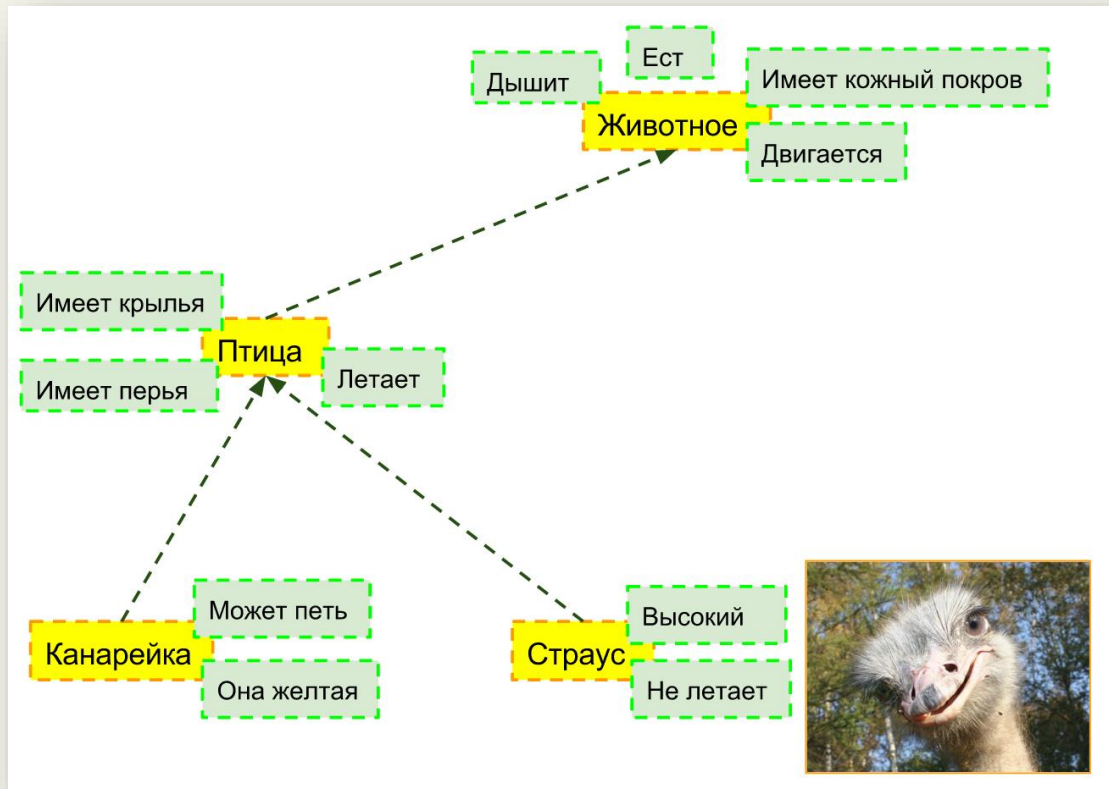
# Посмотрим изменения имени:
print(myDog.getName())
```

Проверим, что все работает:



## 15.2. Наследование в Python

Объектно-ориентированный подход в программировании тесно связан с мышлением человека, с работой его памяти. Для того чтобы нам лучше понять свойства ООП, рассмотрим модель хранения и извлечения информации из памяти человека (модель предложена учеными Коллинзом и Квиллианом)<sup>44</sup>. В своем эксперименте они использовали семантическую сеть, в которой были представлены знания о канарейке:



Например, «канарейка — это желтая птица, которая умеет петь», «птицы имеют перья и крылья, умеют летать» и т. п. Знания в этой сети представлены на различных уровнях: на нижнем уровне располагаются более частные знания, а на верхних — более общие. При таком подходе для понимания высказывания «Канарейка может летать» необходимо воспроизвести информацию о том, что канарейка относится к множеству птиц, и у птиц есть общее свойство «летать», которое распространяется (*наследуется*) и на канареек. Лабораторные эксперименты показали, что реакции людей на простые вопросы типа «Канарейка — это птица?», «Канарейка может летать?» или «Канарейка может петь?» различаются по времени. Ответ на вопрос «Может ли канарейка летать?» требует большего времени, чем на вопрос «Может ли канарейка петь». По мнению Коллинза и Квиллиана, это связано с тем, что информация запоминается человеком на наиболее абстрактном уровне. Вместо того чтобы запоминать все свойства каждой птицы, люди запоминают только отличительные особенности, например, желтый цвет и умение петь у канареек, а все остальные свойства переносятся на более абстрактные уровни: канарейка как птица умеет летать и покрыта перьями; птицы, будучи животными, дышат и питаются и т. д. Действительно, ответ на вопрос «Может ли канарейка дышать?» требует большего времени, т. к. человеку необходимо проследовать по иерархии понятий в своей памяти. С другой стороны, конкретные свойства могут перекрывать более общие, что также требует меньшего времени на обработку информации. Например, вопрос «Может ли страус летать»

<sup>44</sup> см. Гаврилова Т.А., Муромцев Д.И. Интеллектуальные технологии в менеджменте: инструменты и системы

Помимо полей базового класса происходит наследование методов:

```
class Person:
    name = ""
    def __init__(self): # конструктор базового класса
        print("Создан человек")

class Employee(Person):
    job_title = ""

class Customer(Person):
    email = ""

johnSmith = Person()
janeEmployee = Employee()
bobCustomer = Customer()
```

Результат работы программы:

```
>>>
===== RESTART: C:\Python35-32\person.py =====
Создан человек
Создан человек
Создан человек
>>>
```

Таким образом, при создании объектов вызывается конструктор, унаследованный от базового класса. Если дочерние классы содержат собственные методы, то выполняться будут они:

```
class Person:
    name = ""
    def __init__(self): # конструктор базового класса
        print("Создан человек")

class Employee(Person):
    job_title = ""
    def __init__(self): # конструктор дочернего класса
        print("Создан работник")

class Customer(Person):
    email = ""
    def __init__(self): # конструктор дочернего класса
        print("Создан покупатель")

johnSmith = Person()
janeEmployee = Employee()
bobCustomer = Customer()
```

Результат работы программы:

```
>>>
===== RESTART: C:\Python35-32\person.py =====
Создан человек
Создан работник
Создан покупатель
>>>
```

Видим, что в момент создания объекта вызывается конструктор, содержащийся в дочернем классе, т.е. конструктор дочернего класса переопределил конструктор базового класса. Порой требуется вызвать конструктор базового класса из конструктора дочернего класса:

### 15.3. Иерархия наследования в Python

В Python все создаваемые классы наследуются от класса `object`. Создадим класс (собственный тип данных) `Point`, в котором определим (переопределим методы базового класса `object`) специальные методы `__init__`, `__eq__`, `__str__`:

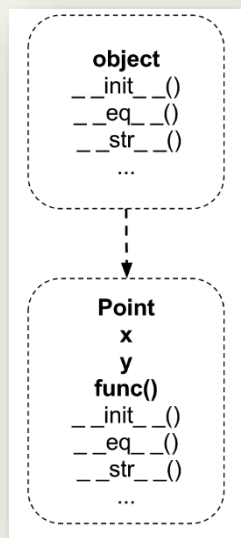
```
class Point:
    def __init__(self, x=0, y=0): # конструктор устанавливает координаты
        self.x = x
        self.y = y
    def __eq__(self, other): # метод для сравнения двух точек
        return self.x == other.x and self.y == other.y
    def __str__(self): # метод для строкового вывода информации
        return "{0.x}, {0.y}".format(self)
    def func(self): # понадобится в следующем примере
        return abs(self.x - self.y)

a = Point() # создаем объект, по умолчанию x=0, y=0
print(str(a)) # здесь вызывается метод __str__ класса Point
# полная форма Point.__str__(a)
b = Point(3, 4)
print(str(b))
b.x = -19
print(a.func())
print(str(b))
print(a == b, a != b) # вызывается метод __eq__
# полная форма для сравнения a == b имеет вид: Point.__eq__(a, b)
```

Результат работы программы:

```
>>>
===== RESTART: C:\Python35-32\point.py =====
(0, 0)
(3, 4)
0
(-19, 4)
False True
>>>
```

Схематично иерархия классов имеет следующий вид:



```
b = Circle(4, 5, 6)
print(str(a)) # здесь вызывается специальный метод __str__()
print(str(b))
print(a == b) # здесь вызывается специальный метод __eq__()
# полная форма вызова метода для a == b: Circle.__eq__(a, b)
print(a == circle)
print(a != circle) # отрицание результата вызова метода __eq__()
# вызов метода базового класса из дочернего называется полиморфизмом:
print(circle.func())
```

Результат работы программы:

```
>>>
===== RESTART: C:\Python35-32\circle.py =====
(4, 5, 6)
(4, 5, 6)
True
False
True
12
>>>
```

Таблицы специальных методов<sup>45</sup>:

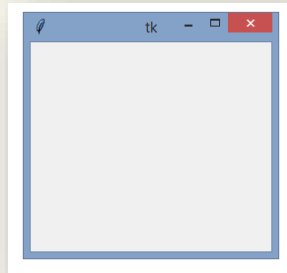
<code>__eq__(self, other)</code>	<code>self == other</code>
<code>__ne__(self, other)</code>	<code>self != other</code>
<code>__lt__(self, other)</code>	<code>self &lt; other</code>
<code>__gt__(self, other)</code>	<code>self &gt; other</code>
<code>__le__(self, other)</code>	<code>self &lt;= other</code>
<code>__ge__(self, other)</code>	<code>self &gt;= other</code>

<code>__add__(self, other)</code>	<code>self + other</code>
<code>__sub__(self, other)</code>	<code>self - other</code>
<code>__mul__(self, other)</code>	<code>self * other</code>
<code>__floordiv__(self, other)</code>	<code>self // other</code>
<code>__truediv__(self, other)</code>	<code>self / other</code>
<code>__mod__(self, other)</code>	<code>self % other</code>
<code>__pow__(self, other)</code>	<code>self ** other</code>

<code>__str__(self)</code>	<code>str(self)</code>
<code>__repr__(self)</code>	<code>repr(self)</code>
<code>__len__(self)</code>	<code>len(self)</code>

<sup>45</sup> <https://docs.python.org/3/reference/datamodel.html#special-method-names>

Результат выполнения программы:



Появилось полноценное окно, которое можно свернуть, растянуть или закрыть! И это только три строчки кода!

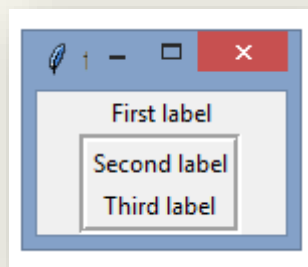
Графические (оконные) приложения отличаются от консольных (без оконных) наличием обработки событий. Для консольных приложений, с которыми мы работали ранее, не требовалось определять, какую кнопку мыши и в какой момент времени нажал пользователь программы. В оконных приложениях важно нажатие мыши, т.к. от этого зависит, например, какой пункт меню выберет пользователь.

Слева на схеме показан алгоритм работы консольной программы, справа – программы с графическим интерфейсом:



Можно изменять параметры фрейма в момент создания объекта<sup>47</sup>:

```
import tkinter
window = tkinter.Tk()
frame = tkinter.Frame(window)
frame.pack()
# Можем изменять параметры фрейма:
frame2 = tkinter.Frame(window, borderwidth=4, relief=tkinter.GROOVE)
frame2.pack()
# Размещаем виджет в первом фрейме (frame)
first = tkinter.Label(frame, text='First label')
first.pack()
# Размещаем виджеты во втором фрейме (frame2)
second = tkinter.Label(frame2, text='Second label')
second.pack()
third = tkinter.Label(frame2, text='Third label')
third.pack()
window.mainloop()
```



В следующем примере для отображения в виджете Label содержимого переменной, используется переменная data класса StringVar (из модуля tkinter). В дальнейшем из примеров станет понятнее, почему в tkinter используются переменные собственного класса<sup>48</sup>.

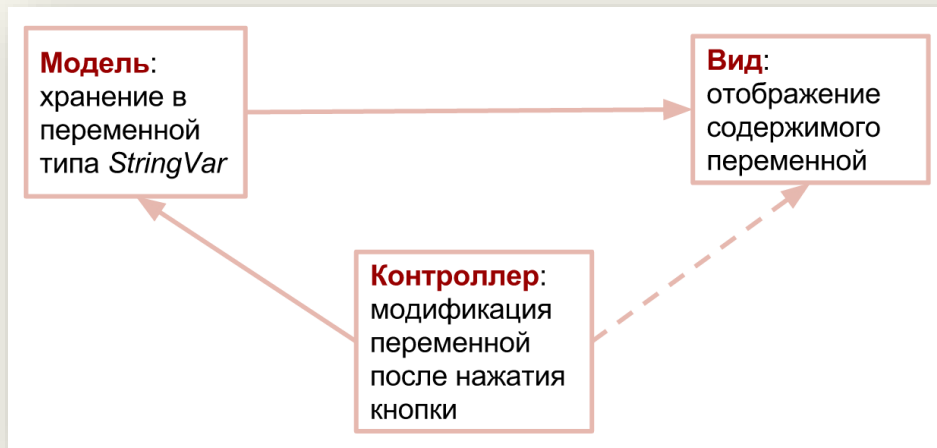
```
import tkinter
window = tkinter.Tk()
# Создаем объект класса StringVar и присваиваем указатель на него data
# (создаем строковую переменную, с которой умеет работать tkinter)
data = tkinter.StringVar()
# Метод set класса StringVar позволяет изменить содержимое переменной:
data.set('Данные в окне')
# textvariable присваиваем ссылку на строковый объект из переменной data
label = tkinter.Label(window, textvariable=data)
label.pack()
window.mainloop()
```

---

<sup>47</sup> <http://effbot.org/tkinterbook/frame.htm>

<sup>48</sup> Tkinter поддерживает работу с переменными классов: BooleanVar, DoubleVar, IntVar, StringVar

Следующая схема показывает связь всех компонентов модели MVC:



Интересная особенность *MVC* в том, что в случае изменения контроллером данных (как это было в предыдущем примере с изменением переменной `var`), «посылается сигнал» *виду* об отображении измененной переменной (перерисовке окна), отсюда получается обновление текста в режиме реального времени.

Следующий пример демонстрирует возможности обработки событий при нажатии на кнопку (виджет `Button`):

```
import tkinter

# Контроллер: функция вызывается в момент нажатия на кнопку
def click():
    # метод get возвращает текущее значение counter
    # метод set устанавливает новое значение counter
    counter.set(counter.get() + 1)

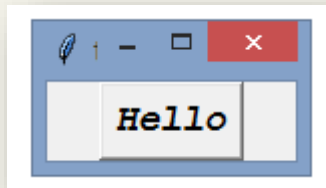
window = tkinter.Tk()
# Модель: создаем объект класса IntVar
counter = tkinter.IntVar()
# Обнуляем созданный объект с помощью метода set
counter.set(0)
frame = tkinter.Frame(window)
frame.pack()
# Создаем кнопку и указываем обработчик (функция click) при нажатии на нее
button = tkinter.Button(frame, text='Click', command=click)
button.pack()
# Вид: в реальном времени обновляется содержимое виджета Label
label = tkinter.Label(frame, textvariable=counter)
label.pack()
window.mainloop()
```

## Изменение параметров по умолчанию при работе с tkinter

Tkinter позволяет изменять параметры виджетов в момент их создания:

```
import tkinter
window = tkinter.Tk()
# Создаем кнопку, изменяем шрифт с помощью кортежа
button = tkinter.Button(window, text='Hello',
                        font=('Courier', 14, 'bold italic'))
button.pack()
window.mainloop()
```

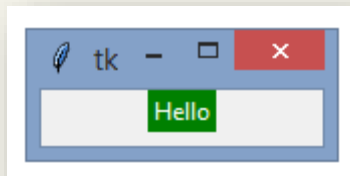
Результат выполнения программы:



В следующем примере изменяются параметры виджета Label:

```
import tkinter
window = tkinter.Tk()
# Изменяем фон, цвет текста:
button = tkinter.Label(window, text='Hello', bg='green', fg='white')
button.pack()
window.mainloop()
```

Результат выполнения программы:

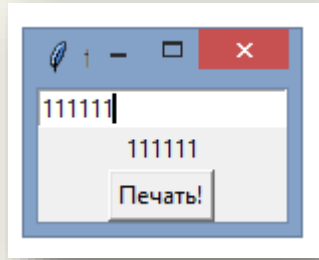


Менеджер расположения (геометрии) pack тоже имеет параметры:

```
import tkinter
window = tkinter.Tk()
frame = tkinter.Frame(window)
frame.pack()
label = tkinter.Label(frame, text='Name')
# Выравнивание по левому краю
label.pack(side='left')
entry = tkinter.Entry(frame)
entry.pack(side='left')
window.mainloop()
```



Результат выполнения программы:



### Упражнение 16.1

Напишите программу, переводящую градусы по Фаренгейту в градусы по Цельсию. Интерфейс работы с программой представлен ниже.



### Упражнение 16.2

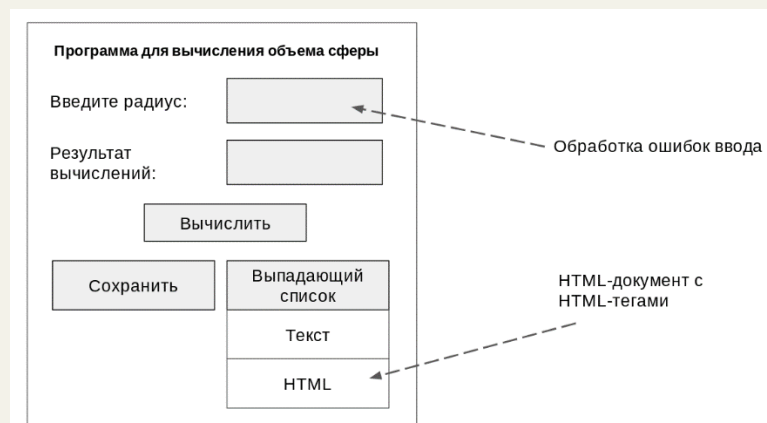
Напишите программу, которая отображает случайное слово на русском языке (тип данных `dict`). Пользователь пытается угадать его на английском (или другом языке). Дополнительно ограничить работу программы по числу неправильно угаданных слов.

### Упражнение 16.3\*

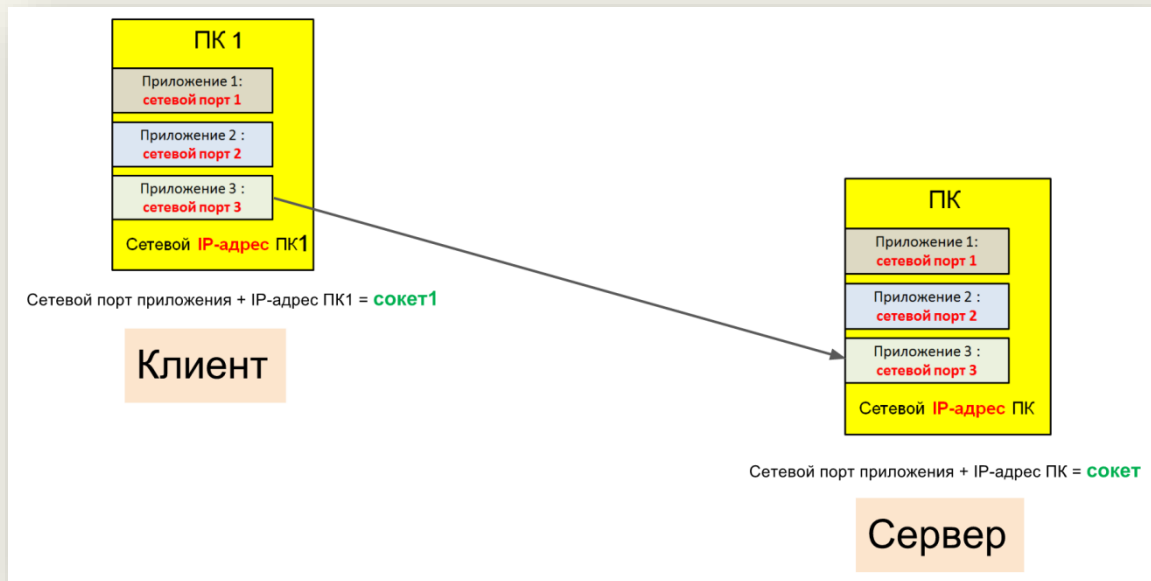
Напишите программу, которая позволяет произвольный текст, введенный с клавиатуры, по нажатию кнопки сохранить в обычный текстовый файл либо в файл HTML-формата (тип файла указывается с помощью выпадающего меню).

### Упражнение 16.4

Разработать программу со следующим графическим интерфейсом, предусмотреть обработку возможных ошибок.



Предположим, что необходимо передать данные от ПК1 (клиента) к ПК (серверу), расположенным в одной сети:



Для идентификации ПК в сети применяются IP-адреса, например, 192.168.0.3. На ПК работает большое число сетевых приложений (Skype, Telegram и пр.), поэтому, чтобы ПК определить, для какого приложения поступили данные, необходимо каждому сетевому приложению присвоить уникальный номер – сетевой порт (например, Skype использует 80 и 443 порты). Связка «IP-адрес, сетевой порт» называется *сокетом* (*socket*). Сокеты предоставляют программный интерфейс для сетевого взаимодействия. Впервые они были реализованы на языке Си в системе BSD. Python имеет встроенный модуль *socket*<sup>51</sup>.

Сетевое взаимодействие происходит посредством клиент-серверного обмена данными, где клиент – запрашивает (отправляет) данные, сервер – обрабатывает данные, полученные от клиента. Например, веб-клиентом является браузер, а веб-сервером – удаленный ПК, способный обрабатывать HTTP-запросы, поступающие от браузера.

Рассмотрим пример серверного и клиентского приложений, написанных на языке Python. Важно, чтобы клиент и сервер запускались в разных экземплярах IDLE, т.е. IDLE необходимо запустить два раза и в отдельном окне сначала запустить программу-сервер, а затем в другом окне запустить программу-клиента.

Клиент-серверное взаимодействие в нашем примере будет происходить на одном и том же ПК, поэтому в качестве IP-адреса указываем 127.0.0.1.

<sup>51</sup> <https://docs.python.org/3/library/socket.html>

Результат работы программы на стороне сервера:

```
>>>
===== RESTART: C:\Python35-32\server.py =====
Connected client
Received[2]: b'Hello world'
Send[3]: b'Hello world'
>>>
```

Результат работы программы на стороне клиента:

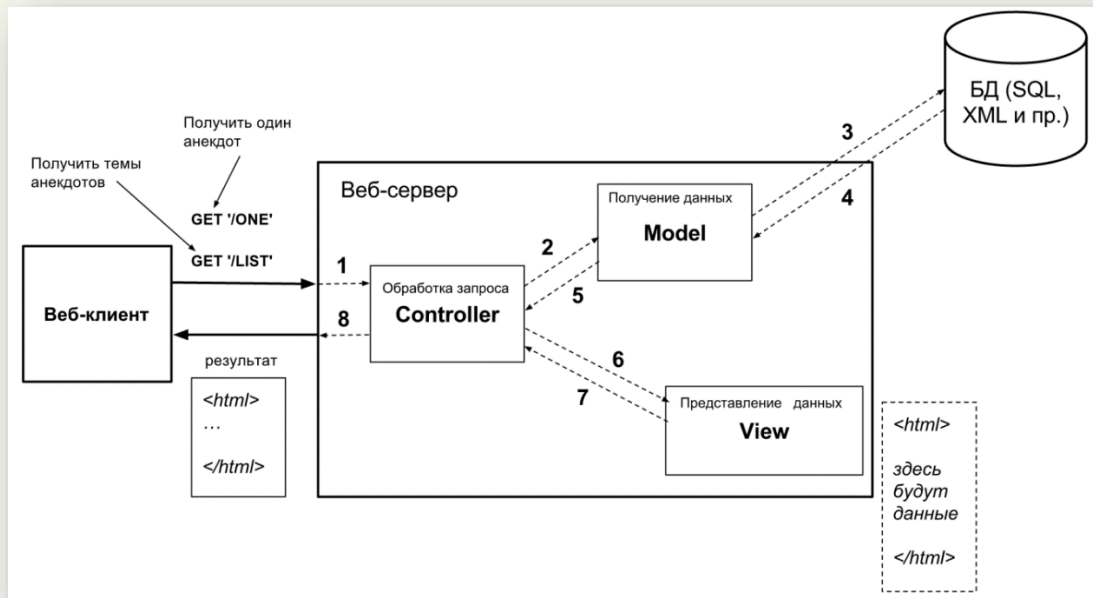
```
>>>
===== RESTART: C:\Python35-32\client.py =====
Send[1]: Hello world
Received[4]: b'Hello world'
>>>
```

### Упражнение 17.1

Разработайте программного бота, работающего по принципу клиент-серверного взаимодействия.

1. Идея бота: переводчик иностранных слов, бот-анекдотов и пр. (можно предлагать собственные идеи).
2. Разработайте систему команд для общения с ботом.
3. Реализацию необходимо построить с использованием шаблона MVC.
4. Оконный интерфейс tkinter.

Рассмотрим схему работы серверного приложения, построенного на основе шаблона MVC (Model-View-Controller):



- 1 - отправка команды от клиента, команда попадает **Контроллеру** – проверка корректности команды и ее обработка;
- 2, 5 - запрос и получение данных от **Модели**;
- 3, 4 - запрос и получение данных из БД, файла и пр.;
- 6, 7 – «сырые» данные отправляются **Виду**, возвращаются данные, имеющие представление (таблица, HTML и пр.);

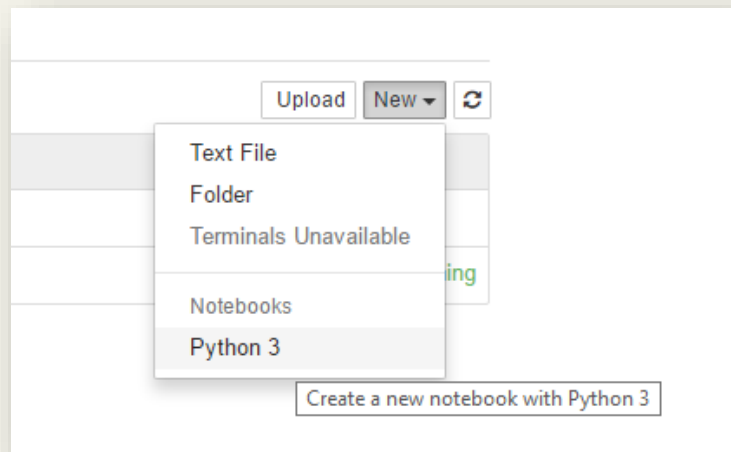
### Упражнение 17.3

Разработайте веб-форму (HTML+PHP) для запроса имени пользователя и пароля из базы данных (MySQL). Пароль состоит из цифр от 1 до 5. Используются GET-запросы. При правильном вводе пароля веб-сервис направляет на страницу, которая содержит «секретную» текстовую строку или ссылку на файл, содержащий «секретную» текстовую строку.

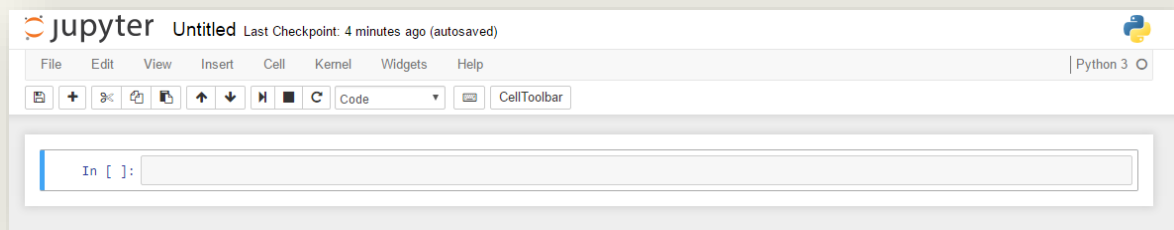
Написать скрипт на языке Python, который создает текстовый файл, содержащий словарь возможных паролей, и на основании созданного словаря перебирает пароли («перебор по словарю») веб-формы. В случае подбора правильного пароля программа считывает и выводит на экран «секретную» текстовую строку.

Построить график зависимости длины пароля от времени перебора.

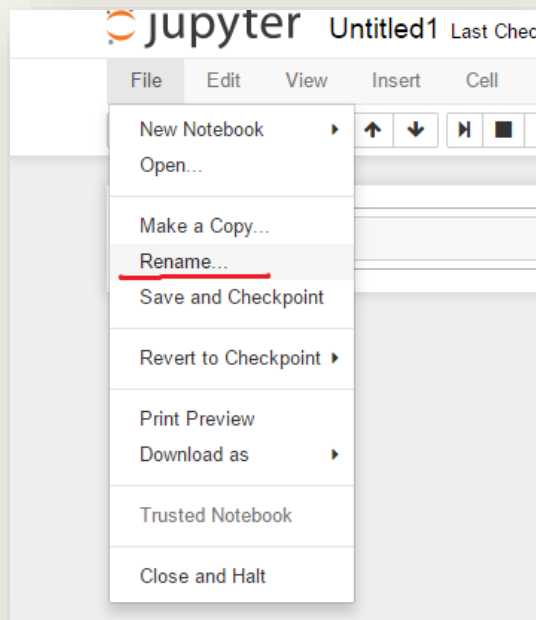
«Python: быстрый старт» <http://dfedorov.spb.ru/python3/>



Откроется веб-интерфейс:

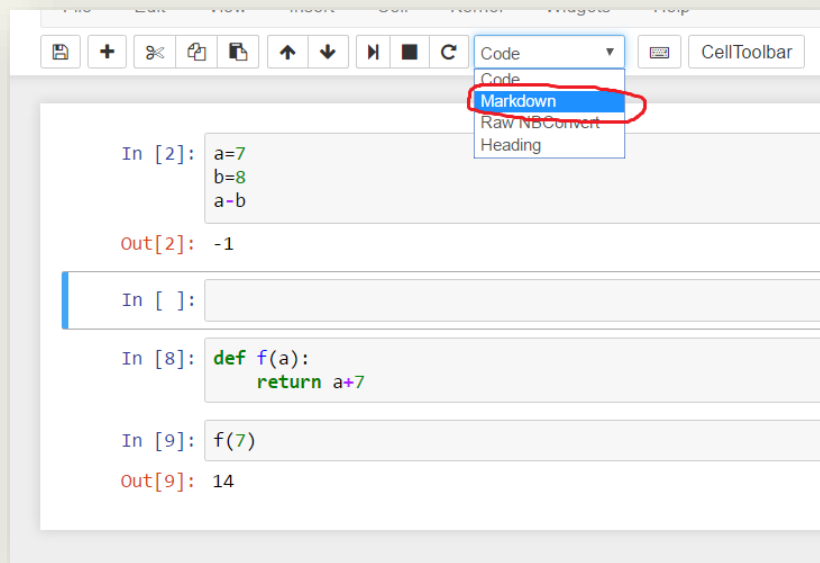


Переименуем блокнот (File → Rename) в MyTest:



Увидим, что в каталоге `\notebooks\` создан файл `MyTest.ipynb`:

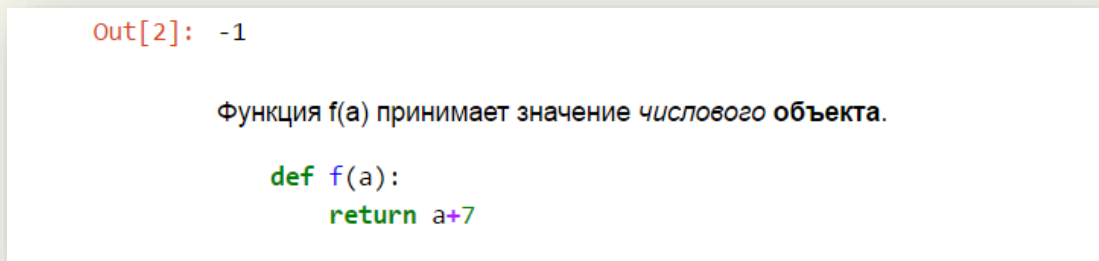
Отдельные ячейки блокнота Jupyter можно отмечать как текстовые (Markdown – специальный язык разметки) для комментирования кода:



Приведем пример разметки:

```
Функция f(a) принимает значение *числового* **объекта**.  
```python  
def f(a):  
    return a+7  
```
```

После заполнения текстовой ячейки ее можно выполнить и язык разметки преобразуется в презентабельный вид:



## 18.2. Работа в Jupyter

В отличие от стандартной среды разработки IDLE Jupyter позволяет:

1. Завершать команды (и пути к файлам) по нажатию клавиши <Tab>.
2. Выводить общую информацию об объекте (*интроспекция объекта*):

Выполним следующий набор команд:


```
In [ ]: lst = [3, 6, 7, 5, 'h', 5]  
lst?
```

В результате получим:

```
In [6]: from IPython.html.widgets import interact

def factorial(x) :
    f = np.math.factorial(x)
    print (str(x) + '!= ' + str(f))

i = interact(factorial , x =(0 ,100))
```



61!= 507580213877224798800856812176625227226004528988036003099405939480985600000000000000

Виджеты работают только при запущенном блокноте Jupyter.

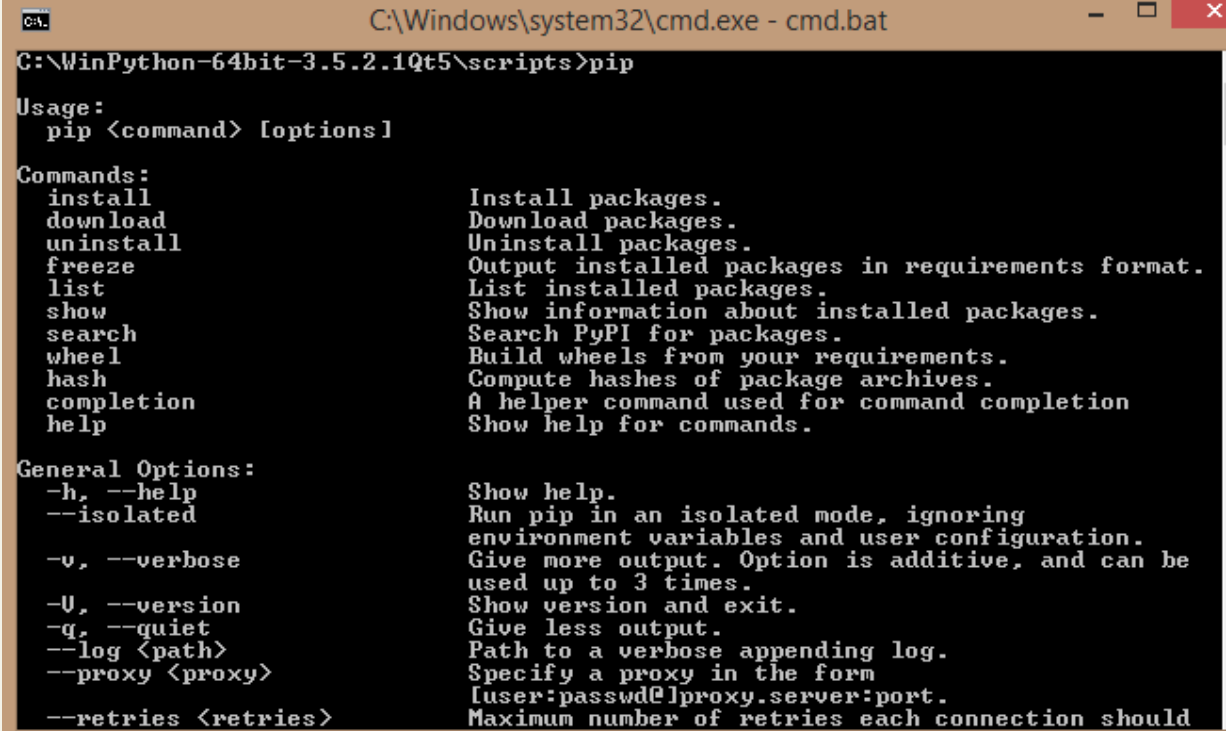
Актуальная версия документации по виджетам<sup>52</sup>: <https://ipywidgets.readthedocs.io/en/latest/>

Различные примеры виджетов:

<http://nbviewer.jupyter.org/github/quantopian/ipython/blob/master/examples/Interactive%20Widgets/Index.ipynb>

#### 18.4. Установка дополнительных пакетов в WinPython из PyPI

Если требуется установить дополнительные пакеты, которые содержатся в PyPI<sup>53</sup>, то запустите WinPython Command Prompt, в появившемся окне наберите команду установки, например, `pip install SPARQLWrapper`:



```
C:\Windows\system32\cmd.exe - cmd.bat
C:\WinPython-64bit-3.5.2.1Qt5\scripts>pip

Usage:
  pip <command> [options]

Commands:
  install           Install packages.
  download          Download packages.
  uninstall         Uninstall packages.
  freeze            Output installed packages in requirements format.
  list              List installed packages.
  show              Show information about installed packages.
  search            Search PyPI for packages.
  wheel             Build wheels from your requirements.
  hash              Compute hashes of package archives.
  completion        A helper command used for command completion
  help              Show help for commands.

General Options:
  -h, --help        Show help.
  --isolated         Run pip in an isolated mode, ignoring
                    environment variables and user configuration.
  -v, --verbose     Give more output. Option is additive, and can be
                    used up to 3 times.
  -U, --version     Show version and exit.
  -q, --quiet       Give less output.
  --log <path>     Path to a verbose appending log.
  --proxy <proxy>   Specify a proxy in the form
                    user:passwd@lproxy.server:port.
  --retries <retries> Maximum number of retries each connection should
```

<sup>52</sup> <https://github.com/ipython/ipywidgets/tree/master>

<sup>53</sup> аббр. от англ. Python Package Index — «каталог пакетов Python»

#### Упражнение 19.4: система «умная страница»<sup>56</sup>

В современном Интернет легко найти сайты, которые адаптируются под пользователя. Поисковики изучают поисковую историю юзера и предлагают более релевантные результаты, сайты-каталоги фильмов умеют подсказать наиболее-интересный видео-контент. Чтобы не отставать от старших товарищей, вам предлагается написать свою систему адаптации контента сайта под пользователя. В качестве критерия адаптации будет использоваться положение пользователя. Необходимо написать сайт, страницы которого (можно сделать всего одну страницу) будут изменяться на основе параметров:

Время суток (утро, день, вечер, ночь)

- Рабочее/не рабочее время
- Рабочий/не рабочий день
- Праздник/не праздник
- Погода (дождь, снег, ясно, пасмурно)
- Город пользователя

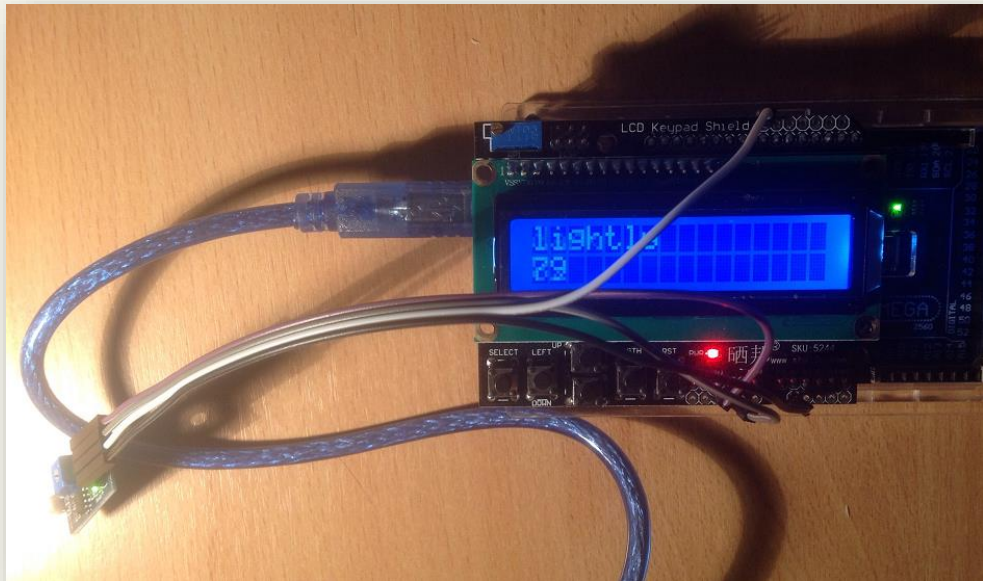
Примеры изменчивого контента:

- Фон сайта (картинки, цвета фона)
- Появляться/исчезать видео
- Изменяться фраза «Добрый <день/вечер/утро/ночь>»
- Сообщать об интересных новостях в городе пользователя

---

<sup>56</sup> Источник задачи: <https://pynsk.ru/tasks/9/>

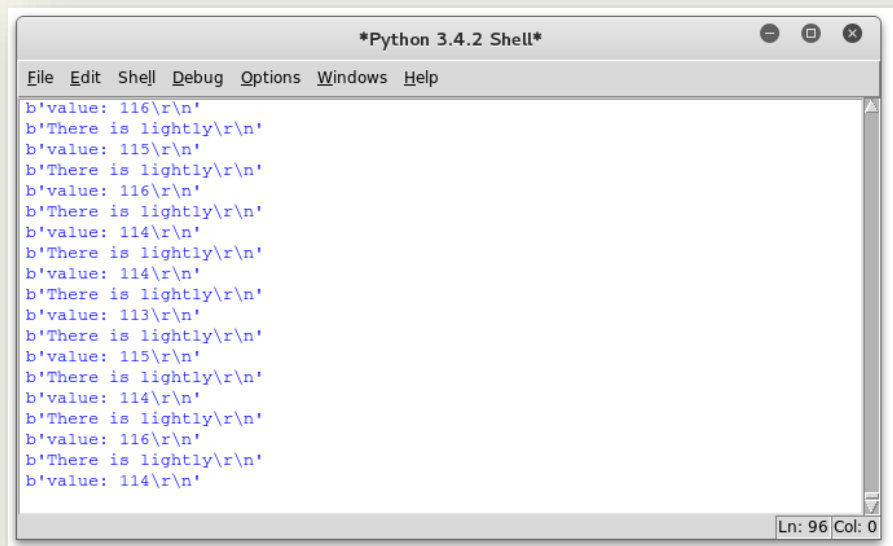




Считать значения, поступающие на последовательный порт, позволяет модуль `pySerial`. Исходный текст на языке Python, считывающий показания, поступающие от датчика освещенности:

```
import serial
s = serial.Serial('/dev/ttyACM0', 9600)
while True:
    print(s.readline())
```

Результат работы программы в режиме реального времени:



Выполняем в командной строке (устанавливаем модуль<sup>61</sup>):

```
# python3 setup.py build
```

```
running build
running build_ext
building 'ownmod' extension
i586-linux-gnu-gcc -pthread -DNDEBUG -g -fwrapv -O2 -Wall -Wstrict-
prototypes -g -fstack-protector-strong -Wformat -Werror=format-
security -D_FORTIFY_SOURCE=2 -fPIC -I/usr/include/python3.4m -c
ownmod.c -o build/temp.linux-i686-3.4/ownmod.o
ownmod.c:23:16: warning: function declaration isn't a prototype [-
Wstrict-prototypes]
    PyMODINIT_FUNC PyInit_ownmod() {
                    ^
ownmod.c: In function 'PyInit_ownmod':
ownmod.c:32:1: warning: control reaches end of non-void function [-
Wreturn-type]
    }
    ^
i586-linux-gnu-gcc -pthread -shared -Wl,-O1 -Wl,-Bsymbolic-functions -
Wl,-z,relro -Wl,-z,relro -g -fstack-protector-strong -Wformat -
Werror=format-security -D_FORTIFY_SOURCE=2 build/temp.linux-i686-
3.4/ownmod.o -o build/lib.linux-i686-3.4/ownmod.cpython-34m.so
```

Выполняем в командной строке с правами администратора:

```
# python3 setup.py install
```

```
running install
running build
running build_ext
running install_lib
copying build/lib.linux-i686-3.4/ownmod.cpython-34m.so ->
/usr/local/lib/python3.4/dist-packages
running install_egg_info
Removing /usr/local/lib/python3.4/dist-packages/ownmod-1.1.egg-info
Writing /usr/local/lib/python3.4/dist-packages/ownmod-1.1.egg-info
```

Теперь можем запустить интерпретатор:

```
# python3.4
```

```
Python 3.4.2 (default, Oct 8 2014, 13:14:40)
[GCC 4.9.1] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import ownmod
>>> ownmod.echo()
вывод из экспортированного кода!
>>>
```

**Официальная документация о расширении и встраивании интерпретатора Python:**

1. Extending and Embedding the Python Interpreter:  
<https://docs.python.org/3.6/extending/index.html>
2. Python/C API Reference Manual: <https://docs.python.org/3.6/c-api/index.html>

---

<sup>61</sup> Подробнее: <https://docs.python.org/3/install/>

## ГЛАВА 23. ОТЗЫВЫ ЧИТАТЕЛЕЙ ОБ ЭЛЕКТРОННОЙ ВЕРСИИ КНИГИ

Присылайте отзывы на почту: [dmitriy.fedoroff@gmail.com](mailto:dmitriy.fedoroff@gmail.com)

Python – один из успешнейших проектов из мира свободного и открытого программного обеспечения (FOSS). Этому языку посвящено целое море литературы как платной, так и бесплатной. Среди общедоступных изданий стоит отметить учебник Д. Ю. Федорова «Основы программирования на примере языка Python».

Прежде всего, достоинства:

### 1. Краткость и ёмкость

Бестселлеры Марка Лутца занимают по 1000+ страниц, что не очень удобно. Зато здесь всего 167 страниц – как раз подойдет для распечатки. Причём это не просто выжимка, а полноценное пособие.

### 2. Композиция

Простая, но стройная композиция:

- \* введение в современное программирование (§ 1-2);

- \* основная часть раскрывается по заявленной во введении схеме: данные + алгоритмы + интерфейс (§ 2-22);

- \* в заключении серия прикладных проектов (§ 23-33).

### 3. Практичность

Теоретические выкладки чередуются со множеством упражнений разной сложности.

### 4. Наглядность

В учебнике масса примеров, иллюстраций, таблиц, блок-схем, шаблонов, а также такой бонус электронных книг (e-book) в отличие от печатных: подсветка синтаксиса.

Недостатки, но они не критичны:

1. Автор часто ссылается на сайт [docs.python.org](http://docs.python.org), но недооценил возможности встроенной документации. Интерактивная справка:

```
>>> help()
```

```
Welcome to Python 3.6's help utility! ...
```

```
help> topics # список тем
```

```
help> STRINGMETHODS # все строковые методы
```

Или напрямую:

```
>>> help("STRINGMETHODS")
```

Даже если нужен непременно веб-интерфейс, то и для этого есть специальная утилита. В командной строке:

```
pydoc3 -b
```

2. Много тем и упражнений со списками (list), но совсем мало со словарями (dict), хотя у них есть такой интересный прием, как «view objects».

3. Раз уж «Python является полностью объектно-ориентированным языком программирования» (с. 116), то саму тему ООП (§ 19) следовало бы осветить более подробно.

Однако время летит. «Десктоп» уже уходит с авансцены ИТ, поэтому айтишникам (да и юзерам тоже) надо обратить внимание на новую платформу – мини-компьютеры, как например, Raspberry Pi. Официальная операционная система Raspbian (GNU/Linux) имеет интегрированную среду разработки Python, причем обеих версий: и 2.7, и 3.

Так что будущее за IoT, и даже здесь «питону» повезло.

*Желаю удачи!*

*Максим Петренко*

## ОБ АВТОРЕ

**Дмитрий Федоров**

Преподаватель [кафедры вычислительных систем и программирования СПбГЭУ](#).

Короткое [деловое резюме](#) и полное [цветное резюме](#)

Мои профили: [персональный блог](#), [РИНЦ](#), [Академия Google](#), [издательство Юрайт](#)

### Области профессиональных интересов:

- разработка и проведение обучающих курсов на основе языка программирования [Python](#);
- проектирование и разработка интеллектуальных систем обучения;
- организация и проведение [трансдисциплинарного межвузовского системного семинара](#);
- исследование [трансформации рынка труда с использованием методов анализа данных](#);
- организация и проведение [групп психолого-педагогической поддержки](#).

**Контакты:** [dmitriy.fedoroff@gmail.com](mailto:dmitriy.fedoroff@gmail.com)

