

Д. Ю. Федоров

ОСНОВЫ
ПРОГРАММИРОВАНИЯ
НА ПРИМЕРЕ ЯЗЫКА
PYTHON

Учебное пособие

Санкт-Петербург
2019

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ.....	4
ГЛАВА 1. ОСНОВЫ ОСНОВ	6
ГЛАВА 2. ЗНАКОМСТВО С ЯЗЫКОМ ПРОГРАММИРОВАНИЯ PYTHON	10
ГЛАВА 3. СТРОКИ И ОПЕРАЦИИ НАД НИМИ	25
ГЛАВА 4. ОПЕРАТОРЫ ОТНОШЕНИЙ	33
ГЛАВА 5. УСЛОВНАЯ ИНСТРУКЦИЯ IF	39
ГЛАВА 6. МОДУЛИ В PYTHON	43
ГЛАВА 7. СОЗДАНИЕ СОБСТВЕННЫХ МОДУЛЕЙ	46
ГЛАВА 8. СТРОКОВЫЕ МЕТОДЫ В PYTHON	52
ГЛАВА 9. СПИСКИ В PYTHON	57
9.1. Создание списка	57
9.2. Операции над списками	59
9.3. Псевдонимы и копирование списков.....	64
9.4. Методы списка	66
9.5. Преобразование типов	67
9.6. Вложенные списки	68
ГЛАВА 10. ИНСТРУКЦИИ ЦИКЛА В PYTHON	70
10.1. Инструкция цикла for.....	70
10.2. Функция range	72
10.3. Подходы к созданию списка.....	74
10.4. Инструкция цикла while.....	78
10.5. Вложенные циклы	81
ГЛАВА 11. ДОПОЛНИТЕЛЬНЫЕ ТИПЫ ДАННЫХ В PYTHON.....	84
11.1. Множества.....	84
11.2. Кортежи	86
11.3. Словари.....	87
ГЛАВА 12. НЕСКОЛЬКО СЛОВ ОБ АЛГОРИТМАХ.....	90
ГЛАВА 13. ОБРАБОТКА ИСКЛЮЧЕНИЙ В PYTHON.....	95
ГЛАВА 14. РАБОТА С ФАЙЛАМИ В PYTHON.....	99
ГЛАВА 15. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ В PYTHON	108
15.1. Основы объектно-ориентированного подхода	108
15.2. Наследование в Python	114
15.3. Иерархия наследования в Python	118
ГЛАВА 16. РАЗРАБОТКА ПРИЛОЖЕНИЙ С ГРАФИЧЕСКИМ ИНТЕРФЕЙСОМ.....	121
16.1. Основы работы с модулем tkinter	121
16.2. Шаблон «Модель-вид-контроллер» на примере модуля tkinter.....	125
ГЛАВА 17. КЛИЕНТ-СЕРВЕРНОЕ ПРОГРАММИРОВАНИЕ В PYTHON	131
ГЛАВА 18. СРЕДА РАЗРАБОТКИ JUPYTER	137
18.1. Установка и запуск Jupyter	137
18.2. Работа в Jupyter	140
18.3. Интерактивные виджеты в Jupyter Notebook	141
18.4. Установка дополнительных пакетов в WinPython из PyPI	142
ГЛАВА 19. ПРИМЕНЕНИЕ ЯЗЫКА PYTHON	143
19.1. В области защиты информации и системного администрирования.....	143
19.2. В области искусственного интеллекта	143
ГЛАВА 20. ПРОГРАММИРОВАНИЕ КОНТРОЛЛЕРА ARDUINO	145
ГЛАВА 21. ИМПОРТИРОВАНИЕ МОДУЛЕЙ, НАПИСАННЫХ НА ЯЗЫКЕ С	147
ГЛАВА 22. ДОПОЛНИТЕЛЬНЫЕ МАТЕРИАЛЫ ПО ЯЗЫКУ PYTHON	149
ГЛАВА 23. ОТЗЫВЫ ЧИТАТЕЛЕЙ ОБ ЭЛЕКТРОННОЙ ВЕРСИИ КНИГИ.....	150
ОБ АВТОРЕ	152

Д. Ю. Федоров. «Основы программирования на примере языка Python»

Этот учебник посвящается памяти:



Андрея Петровича Ершова
(19 апреля 1931 г. - 8 декабря 1988 г.)

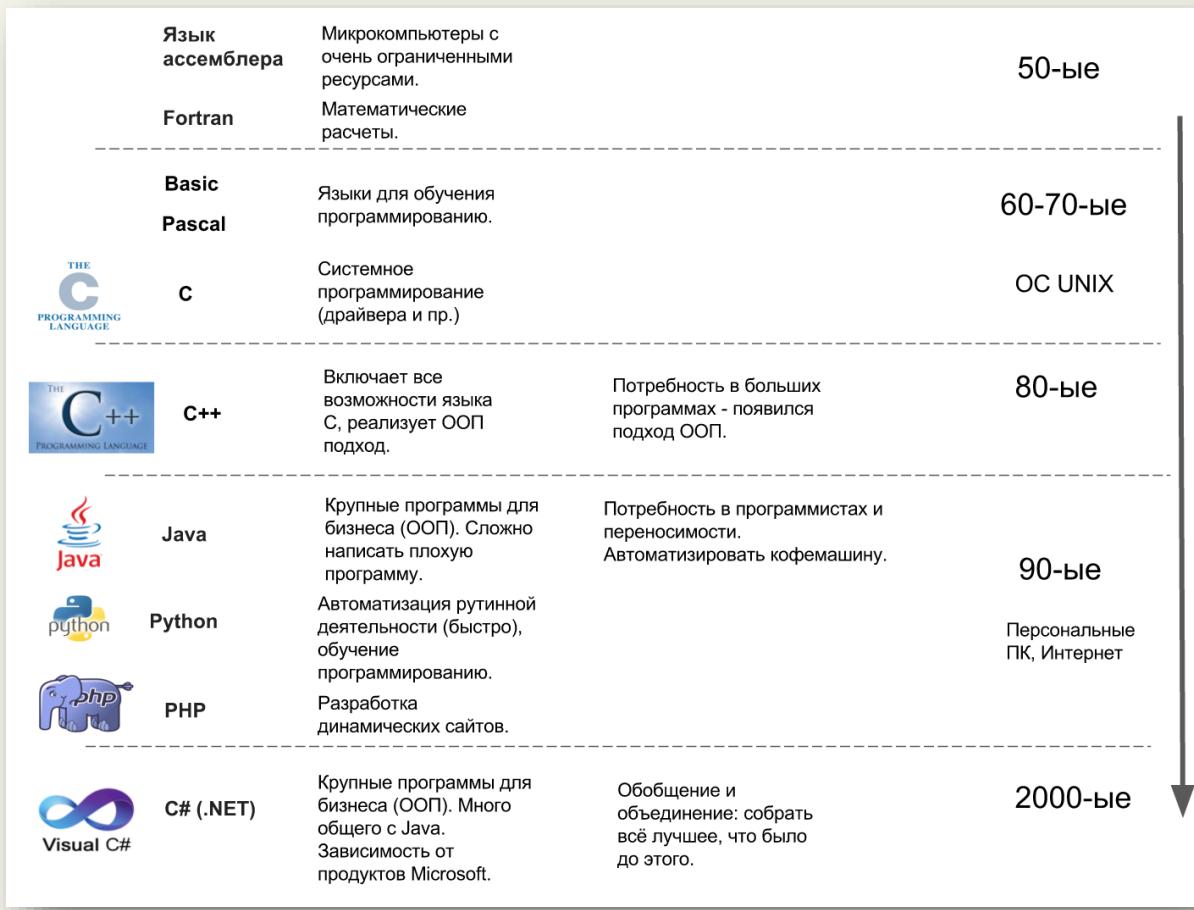
Советский ученый, один из пионеров теоретического и системного программирования, создатель Сибирской школы информатики, академик АН СССР. В 80-х годах прошлого века начал эксперименты по преподаванию программирования в средней школе, которые привели к введению курса информатики в средние школы страны.



Сеймура Пейпerta
(29 февраля 1928 г. - 31 июля 2016 г.)

Математик, программист, психолог и педагог. Один из основоположников теории искусственного интеллекта, создатель языка LOGO.

У языков программирования интересная история. Они создавались не на пустом месте, а под конкретные задачи, стоявшие на тот момент перед их разработчиками, отсюда становится понятной область применения того или иного языка программирования. На сегодняшний день существуют тысячи языков программирования, но наибольшую роль сыграли лишь некоторые из них.



Язык ассемблера	Микрокомпьютеры с очень ограниченными ресурсами.	50-ые	
Fortran	Математические расчеты.		
Basic	Языки для обучения программированию.	60-70-ые	
Pascal			
C	Системное программирование (драйвера и пр.)	ОС UNIX	
C++	Включает все возможности языка C, реализует ООП подход.	80-ые	
Java	Крупные программы для бизнеса (ООП). Сложно написать плохую программу.	Потребность в больших программах - появился подход ООП.	90-ые
Python	Автоматизация рутинной деятельности (быстро), обучение программированию.	Потребность в програмистах и переносимости. Автоматизировать кофемашину.	Персональные ПК, Интернет
PHP	Разработка динамических сайтов.		
C# (.NET)	Крупные программы для бизнеса (ООП). Много общего с Java. Зависимость от продуктов Microsoft.	Обобщение и объединение: собрать всё лучшее, что было до этого.	2000-ые
Visual C#			

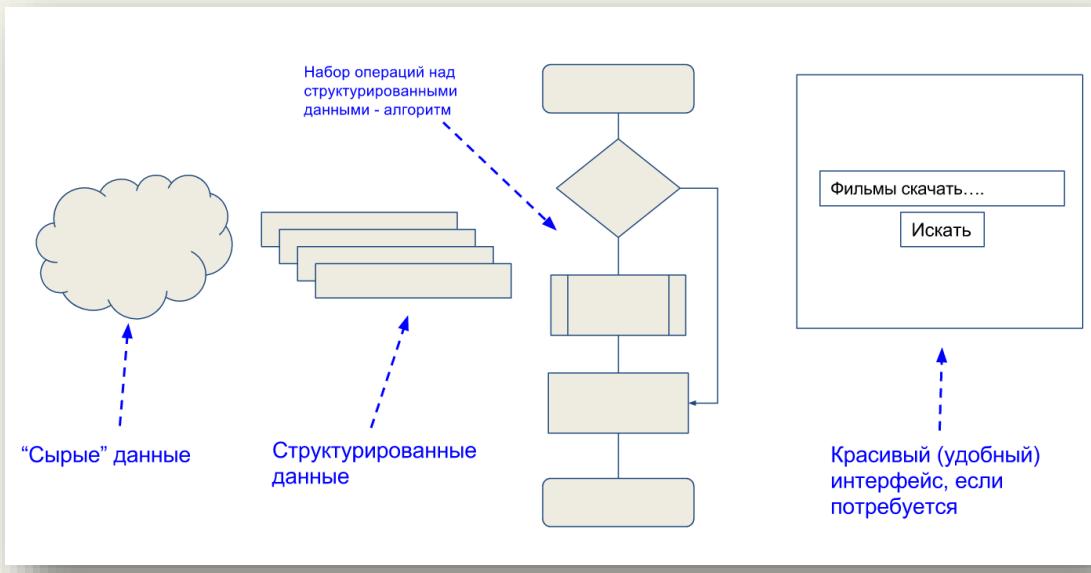
Ранее мы сказали, что началом общения с компьютером послужил машинный код. Затем в 50-ые годы двадцатого века появился низкоуровневый язык ассемблера, наиболее приближенный к машинному уровню. Он привязан к процессору, поэтому его изучение равносильно изучению архитектуры процессора. На языке ассемблера пишут программы и сегодня, он незаменим в случае небольших устройств (микроконтроллеров), обладающих очень ограниченными ресурсами памяти.

Следующий этап – появление языка Фортран, предназначавшегося для математических вычислений.

Со временем росла потребность в новых кадрах и необходимость в обучении программированию. Обучение на языках ассемблера или Фортране требовало много сил, поэтому в 60-70-ые годы появляется плеяда языков для обучения: Basic, Pascal. Язык Pascal до сих пор используется в школах в качестве основного языка обучения программированию.

В это же время ведутся исследования в области разработки операционных систем, что приводит к появлению системы UNIX. Первоначально эта операционная система была написана на языке ассемблера, что усложняло ее модификацию и изучение, тогда Д. Ритчи разработал язык С для системного программирования и совместно с Б. Керниганом переписал систему UNIX на этом языке. Впоследствии операционная система UNIX получила широкое распространение (в наши дни больше известны ее клоны GNU/Linux), а

Так что же такое программа и какие шаги требуется выполнить для ее написания?



На первом шаге у программиста есть набор «сырых» данных. Это, к примеру, могут быть разрозненные бухгалтерские отчеты, статистика и пр. Эти сведения необходимо структурировать и поместить в компьютер. Сравним написание программы с приготовлением салата: есть «сырые» овощи, которые нужно помыть и порезать, т.е. структурировать.

Затем, если задачу можно разбить на отдельные небольшие подзадачи, то лучше так и поступить. Решить небольшие задачи, убедиться, что они работают и объединить их обратно. На научном языке это называется анализом и синтезом. С опытом приходит умение видеть и выделять подзадачи.

Далее, программистом реализуется алгоритм, т.е. набор действий для обработки структурированных данных, исходя из поставленной задачи. Отмечу, что правильный выбор структуры данных влияет на создание (выбор) алгоритма. Мощь языка программирования отчасти заключена в структурах данных, которое он предоставляет для работы.

После того, как алгоритм разработан и программа работает (в результате ее работы получается корректный ответ), можно создавать красивый и удобный интерфейс. Часто сталкиваюсь с мнением, что визуальные среды способствуют изучению программированию. Не соглашусь с этим, т.к. визуальная среда становится доминирующей и много сил уходит на ее изучение, вместо того, чтобы заниматься главным (структурой и алгоритмизацией). Посмотрите на сайт поисковой системы – поле для ввода с одной кнопкой. Простота скрывает за собой сложные интеллектуальные алгоритмы, которые работают на стороне сервера.

Исходя из рассмотренного алгоритма разработки программы, мы построим наш курс. Начнем с изучения структур данных, добавим алгоритмы, а завершим созданием графического интерфейса.

программу на Python в системе Windows, ее можно запустить, например, в GNU/Linux и получить такой же результат.

Скачать и установить² интерпретатор Python можно совершенно бесплатно с официального сайта: <http://python.org>. Для работы нам понадобится интерпретатор **Python версии 3** или выше³.

После установки программы запустите интерактивную графическую среду IDLE и дождитесь появления приглашения для ввода команд:

```
Type "copyright", "credits" or "license()" for more information.  
>>>
```

В самом начале обучения Python можно представить как обычный интерактивный калькулятор. В интерактивном режиме IDLE найдем значения следующих математических выражений⁴. После завершения набора выражения нажмите клавишу **Enter** для завершения ввода и вывода результата на экран.

```
>>> 3.0 + 6  
9.0  
>>> 4 + 9  
13  
>>> 1 - 5  
-4  
>>> _ + 6  
2  
>>>
```

Нижним подчеркиванием в предыдущем примере обозначается последний полученный результат.

Если по какой-либо причине совершил ошибку при вводе команды, то Python сообщит об этом:

```
>>> a  
Traceback (most recent call last):  
  File "<pyshell#0>", line 1, in <module>  
    a  
NameError: name 'a' is not defined  
>>>
```

Не бойтесь совершать ошибки! Python поправит и подскажет, на что следует обратить внимание.

² Для обучения в ОС Linux понадобится установить редактор IDLE: sudo apt-get install idle3

³ Чем выше версия, тем лучше.

⁴ Числа могут быть представлены в различных системах счисления:

```
>>> 0b10 # по основанию 2  
2  
>>> 0o10 # по основанию 8  
8  
>>> 0x10 # по основанию 16  
16
```

Рассмотрим выражение $y=x+3*6$, где y и x являются переменными, которые могут содержать значения числового типа. На языке Python вычислить значение y при x равном 1 можно следующим образом:

```
>>> x = 1
>>> y = x + 3*6
>>> y
19
>>>
```

В выражении нельзя использовать переменную, если ранее ей не было присвоено значение - для Python такие переменные не определены.

Содержимое переменной y можно увидеть, если в интерактивном режиме набрать ее имя.

Имена переменным придумывает программист, но есть несколько ограничений, связанных с наименованием. В качестве имен переменных нельзя использовать ключевые слова, которые для Python имеют определенный смысл (эти слова подсвечиваются в IDLE оранжевым цветом):

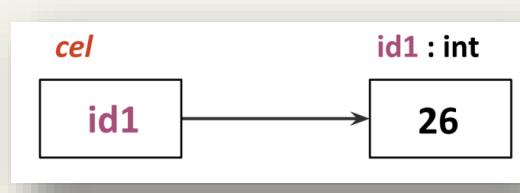
and	as	assert	break	class	continue
def	del	elif	else	except	
finally	for	from	global	if	import
in	is	lambda	nonlocal	not	or
pass	raise	return	try	while	with
yield	True	False	None		

Далее мы часто будем обращаться к формуле перевода из шкалы в градусах по Цельсию в шкалу градусов по Фаренгейту и обратно. Формула перевода из градусов по Цельсию (T_C) в градусы по Фаренгейту (T_F) имеет вид:

$$T_F = 9/5 * T_C + 32$$

Найдем значение T_F при T_C равном 26. Создадим переменную с именем `cel`, содержащую значение целочисленного типа 26.

```
>>> cel = 26
>>> cel
26
>>> 9/5 * cel + 32
78.80000000000001
>>>
```



Остановимся подробно на том, как Python работает с переменными. Здесь есть существенная особенность, которая отличает его от других языков программирования.

Ранее мы сказали, что Python – объектно-ориентированный язык программирования. В чем это выражается?

новые¹⁰. Это особенность числового типа данных – объекты этого типа являются неизменяемыми.

У начинающих программистов часто возникает недоумение при виде следующих вычислений:

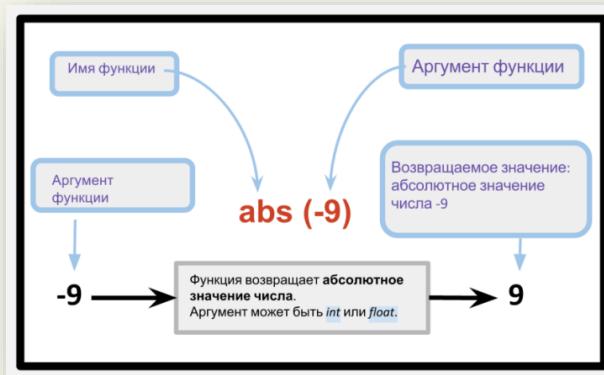
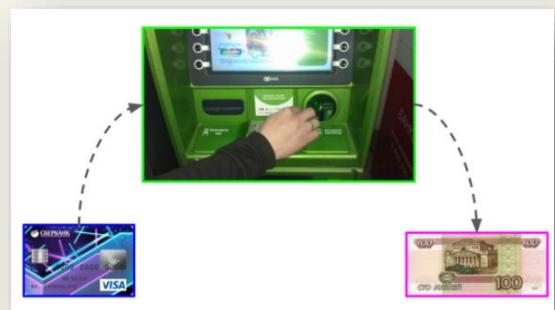
```
>>> num = 20
>>> num = num * 3 # сокращенно: num *= 3
>>> num
60
>>>
```

Если вспомнить, что сначала вычисляется правая часть, то все легко объясняется.

Функция в Python является основой при написании программ. С чем можно сравнить функцию? Напрашивается аналогия с «черным ящиком», когда мы знаем, что поступает на вход и что при этом получается на выходе, а внутренности «черного ящика» часто бывают от нас скрыты. Примером является банкомат.

На вход банкомата поступает пластиковая карточка (пин-код, денежная сумма), на выходе мы ожидаем получить запрашиваемую сумму. Нас не очень сильно интересует принцип работы банкомата до тех пор, пока он работает без сбоев.

Рассмотрим функцию с именем `abs`, принимающую на вход один аргумент – объект числового типа и возвращающую абсолютное значение для этого объекта.



Пример вызова функции `abs` с аргументом `-9` имеет вид:

```
>>> abs (-9)
9
>>> d = 1
>>> n = 3
>>> abs(d - n)
2
```

¹⁰ Информация для опытных программистов. Для экономии ресурсов при работе с небольшими целыми значениями Python ссылается на уже существующие в памяти объекты

```
>>> int(5.6)
5
>>> int()
0
>>> float(5)
5.0
>>> float()
0.0
>>>
```

В качестве упражнения найдите значения следующих выражений:

```
pow(abs(-5) + abs(-3), round(5.8))
int(round(pow(round(5.777, 2), abs(-2)), 1))
```

Откуда брать описание работы функций? Программисты для этого используют документацию. В Python документация для функции может быть вызвана с помощью функции `help`, на вход которой передается имя функции:

```
>>> help(abs)
Help on built-in function abs in module builtins:

abs(x, /)
    Return the absolute value of the argument.

>>>
```

Вернемся к формуле перевода градусов по шкале Фаренгейта (T_F) в градусы по шкале Цельсия (T_C):

$$T_C = \frac{5}{9} * (T_F - 32)$$

Произведем несколько вычислений с использованием Python, где переменная `deg_f` будет содержать значение в градусах по Фаренгейту:

```
>>> deg_f = 80
>>> deg_f
80
>>> 5/9 * (deg_f - 32)
26.666
>>> deg_f = 70
>>> 5/9 * (deg_f - 32)
```

Заметим, что каждый раз нам приходится набирать одну и ту же формулу для перевода. Упростим наши вычисления, создав собственную функцию, переводящую градусы по Фаренгейту в градусы по Цельсию.

```
>>> def convert_co_cels(fahren) :  
    return (fahren-32)*5/9  
  
>>> convert_co_cels(451)  
232.7777777777777  
>>> convert_co_cels(300)  
148.8888888888889  
>>>
```

После того как функция создана, можно ее вызывать, передавая в скобках различные аргументы.

Для закрепления создайте собственные функции для вычисления следующих выражений:

$$\begin{aligned}x^4 + 4^x \\y^4 + 4^x\end{aligned}$$

Внимательный читатель заметил, что в интерактивном режиме нельзя внести изменения в выражение, которое уже ранее было выполнено. Приходится повторно набирать выражение и его запускать. В случае больших программ удобно использовать отдельные файлы с расширением .py.

В меню IDLE выберете File → New File. Появится окно текстового редактора, в котором можно набирать команды на языке Python.

Наберем следующий код:

```
a=5  
print(a)  
print(a+5)
```

В меню редактора выберем Save As и сохраним файл в произвольную директорию, указав имя myprog1.py. В старых версиях IDLE приходится вручную прописывать расширение у файла.

Чтобы выполнить программу в меню редактора выберем Run -> Run Module (или нажмем <F5>). Результат работы программы отобразится в интерактивном режиме (у меня получилось так):

```
>>>  
===== RESTART: C:/Python35-32/myprog1.py =====  
5  
10  
>>>
```

Здесь нам следует познакомиться с функцией print, которая отображает содержимое переменных, переданных ей в качестве аргументов. Вспомните, что в интерактивном режиме мы просто набирали имя переменной, что приводило к выводу на экран ее содержимого. Дело в том, что Python в интерактивном режиме самостоятельно подставляет вызов функции print, а в файле нам придется делать это вручную.

Разберемся теперь, как создавать функции в отдельном файле и вызывать их.

Рассмотрим пример. В отдельный файл с именем `myprog.py` поместим следующий код:

```
a = 3 # глобальная переменная
print('глобальная переменная a = ', a)
y = 8 # глобальная переменная
print('глобальная переменная y = ', y)

def func():
    print('func: глобальная переменная a = ', a)
    y = 5 # локальная переменная
    print('func: локальная переменная y = ', y)

func() # вызываем функцию func
print('??? y = ', y) # отобразится глобальная переменная
```

Обращаю внимание, что у функции `print` могут быть несколько аргументов, заданных через запятую. В одинарные кавычки помещается строка¹⁴.

После выполнения программы получим следующий результат:

```
>>>
=====
RESTART: C:/Python35-32/myprog.py =====
глобальная переменная a = 3
глобальная переменная y = 8
func: глобальная переменная a = 3
func: локальная переменная y = 5
??? y = 8
>>>
```

Внутри функции мы смогли обратиться к глобальной переменной `a` и вывести ее значение на экран. Далее внутри функции создается локальная переменная `y`, причем ее имя совпадает с именем глобальной переменной – в этом случае при обращении к `y` выводится содержимое локальной переменной, а глобальная остается неизменной.

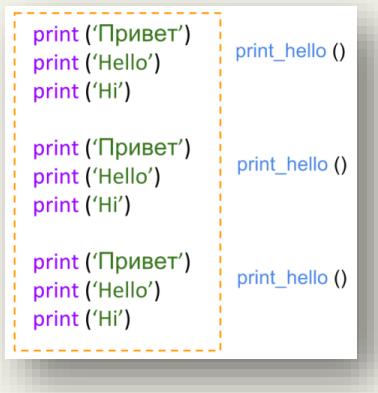
Как быть, если мы хотим изменить содержимое глобальной переменной внутри функции? Ниже показан пример такого изменения с использованием ключевого слова `global`¹⁵:

```
x = 50 # глобальная переменная
def func():
    global x # указываем, что x - глобальная переменная
    print('x равно', x)
    x = 2    # изменяем глобальную переменную
    print('Заменяем глобальное значение x на', x)

func()
print('Значение x составляет', x)
```

¹⁴ О строках подробно поговорим в следующей главе

¹⁵ В Python «явное лучше неявного».



Упражнение 2.1

Создайте в отдельном файле функцию, переводящую градусы по шкале Цельсия в градусы по шкале Фаренгейта по формуле: $T_F = 9/5 * T_C + 32$

Упражнение 2.2

Создайте в отдельном файле функции, вычисляющие площадь и периметр квадрата.

Упражнение 2.3

Напишите функцию в отдельном файле, вычисляющую среднее арифметическое трех чисел.

Для справки. Функции в Python

Здесь и далее в п. «Для справки» приводятся более сложные примеры использования # языка Python. Если при первом прочтении возникли трудности, то рекомендую # вернуться к этому разделу позже.

Рассмотрим несколько полезных особенностей при работе с функциями в Python. Имена функций в Python являются переменными, содержащими адрес объекта¹⁷ типа функция¹⁸, поэтому этот адрес можно присвоить другой переменной и вызвать функцию с другим именем.

```
def summa(x, y):
    return x + y
f = summa
v = f(10, 3) # вызываем функцию с другим именем
```

Параметры функции могут принимать значения по умолчанию:

```
def summa(x, y=2):
    return x + y
a = summa(3) # вместо y подставляется значение по умолчанию
b = summa(10, 40) # теперь значение второго параметра равно 40
```

Ранее мы сказали, что имя функции – обычная переменная, поэтому можем передать ее в качестве аргумента при вызове функции:

¹⁷ В Python все является объектами.

¹⁸ Да, да, это еще один тип данных.

ГЛАВА 3. СТРОКИ И ОПЕРАЦИИ НАД НИМИ

Python часто используют для обработки текстов: поиска в тексте, замены отдельных частей текста и т.д. Для работы с текстом в Python предусмотрен специальный строковый тип данных `str`.

Python создает строковые объекты, если текст поместить в одинарные или двойные кавычки:

```
>>> 'hello'  
'hello'  
>>> "Hello"  
'Hello'  
>>>
```

Без кавычек Python расценит текст как переменную и попытается вывести на экран ее содержимое (если такая переменная была создана):

```
>>> hello  
Traceback (most recent call last):  
  File "<pyshell#2>", line 1, in <module>  
    hello  
NameError: name 'hello' is not defined  
>>>
```

Можно создать пустую строку:

```
>>> ''  
''  
>>>
```

Для работы со строками в Python предусмотрено большое число встроенных функций, например, `len`. Эта функция определяет длину строки, которая передается ей в качестве аргумента.

```
>>> help(len)  
Help on built-in function len in module builtins:  
  
len(obj, /)  
    Return the number of items in a container.  
  
>>> len('Привет!')  
7  
>>>
```

К примеру, если мы хотим объединить несколько строк в одну, Python позволяет это сделать с помощью операции конкатенации (обычный символ `+` для строк):

```
>>> 'Привет, ' + 'земляне!'  
'Привет, земляне!'  
>>>
```

Первый – заключить в кавычки разных типов, чтобы Python понял, где заканчивается строка.

Второй – использовать специальные символы (управляющие escape-последовательности), которые записываются, как два символа, но Python видит их как один:

```
>>> len("\\")  
1  
>>>
```

Полезно знать об этих символах, т.к. они часто используются при работе со строками:

\n	- перевод на новую строку
\t	- знак табуляции
\\\	- наклонная черта влево
\'	- символ одиночной кавычки
\"	- символ двойной кавычки

При попытке перенести длинную строку на новую:

```
>>> 'Это длинная  
SyntaxError: EOL while scanning string literal  
>>> строка  
Traceback (most recent call last):  
  File "<pyshell#20>", line 1, in <module>  
    строка  
NameError: name 'строка' is not defined  
>>>
```

Создадим многострочную строку (необходимо заключить ее в три одинарные кавычки):

```
>>> '''Это длинная  
строка'''  
'Это длинная\nстрока'  
>>>
```

При выводе на экран перенос строки отобразился в виде специального символа '\n'.

Ранее мы говорили о функции `print`, которая отображает на экране объекты разных типов данных, передаваемых ей в качестве входных аргументов. Теперь снова к ней вернемся. Передадим на вход функции `print` строку со специальным символом:

```
>>> print('Это длинная\nстрока')  
Это длинная  
строка  
>>>
```

Функция `print` специальный символ смогла распознать и сделать перевод строки.

В примере мы вызвали функцию `input` и результат ее работы присвоили переменной `s`. Далее пользователь ввел значение с клавиатуры и нажал Enter, т.е. закончил ввод. Содержимое переменной вывели на экран. Вызвали функцию `type`, которая позволяет определить тип объекта. Увидели, что это строка.



И здесь **ВНИМАНИЕ**: не забывайте, что функция `input` возвращает строковый объект!



Вот, чем это может обернуться:

```
>>> s = input("Введите число: ")
Введите число: 555
>>> s+5
Traceback (most recent call last):
  File "<pyshell#33>", line 1, in <module>
    s+5
TypeError: Can't convert 'int' object to str implicitly
>>>
```

В этом примере используется входной аргумент функции `input`, который выведет строку-приглашение перед пользовательским вводом: "Введите число: ".

После ввода значения с клавиатуры мы пытаемся его сложить с числом 5, а вместо ожидаемого результата получаем сообщение об ошибке. Использование операции `+` для строкового и числового объектов привело Python в ступор. Как решить проблему? Преобразованием типов:

```
>>> s = int(input("Введите число: "))
Введите число: 555
>>> s + 5
560
>>>
```

Теперь все получилось! Будьте внимательны!

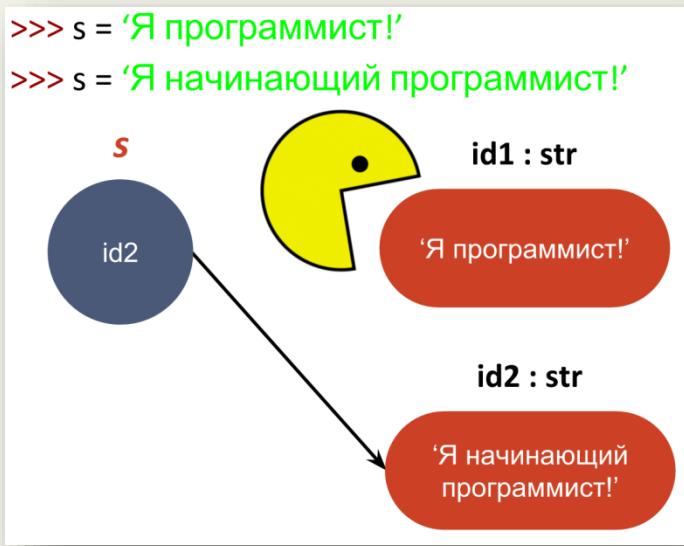
Упражнение 3.1

Попросите пользователя ввести свое имя и после этого отобразите на экране строку вида: Привет, <имя>! Вместо <имя> должно указываться то, что пользователь ввел с клавиатуры.

```
Как тебя зовут?
Вася
Привет, Вася!
```

Со строками познакомились, научились их создавать. Теперь рассмотрим операции над ними.

Изменяем значение переменной `s`. Создается новый строковый объект (а не изменяется предыдущий) по адресу `id2` и этот адрес записывается в переменную `s`.



Прежде чем мы поймем, как строки можно изменять, познакомимся со срезами:

```
>>> s = 'Питоны водятся в Африке'  
>>> s[1:3]  
'ит'  
>>>
```

`s[1:3]` – срез строки `s`, начиная с индекса 1, заканчивая индексом 3 (не включительно). Это легко запомнить, если индексы представить в виде смещений:



Со срезами можно производить различные манипуляции:

```
>>> s[:3] # с 0 индекса по 3-ий не включительно  
'Пит'  
>>> s[:] # вся строка  
'Питоны водятся в Африке'  
>>> s[::-2] # третий аргумент задает шаг (по умолчанию один)  
'Птн ояс фие'  
>>> s[::-1] # «обратный» шаг  
'екирфа в ястядов ынотип'  
>>> s[:-1] # вспомним, как мы находили отрицательный индекс  
'Питоны водятся в Африк'
```

ГЛАВА 4. ОПЕРАТОРЫ ОТНОШЕНИЙ

Числа можно сравнивать. В Python для этого есть следующие операции сравнения:

>	Больше
<	Меньше
>=	Больше или равно
<=	Меньше или равно
==	Равно
!=	Не равно

В интерактивном режиме сравним два числа:

```
>>> 6 > 5
True
>>> 7 < 1
False
>>> 7 == 7      # не путайте == и =
True
>>> 7 != 7
False
>>>
```

Python возвращает `True`¹⁹ (Истина == 1²⁰), когда сравнение верное и `False` (Ложь == 0) – в ином случае. `True` и `False` относятся к логическому (булевому) типу данных `bool`.

```
>>> type(True)
<class 'bool'>
>>>
```

Отдельного разговора заслуживает сравнение вещественных чисел, т.к. оно может привести к неожиданным, на первый взгляд, результатам (см. стандарт IEEE 754):

```
>>> 0.1 + 0.1 == 0.2
True
>>> 0.1 + 0.1 + 0.1 == 0.3
False
>>>
```

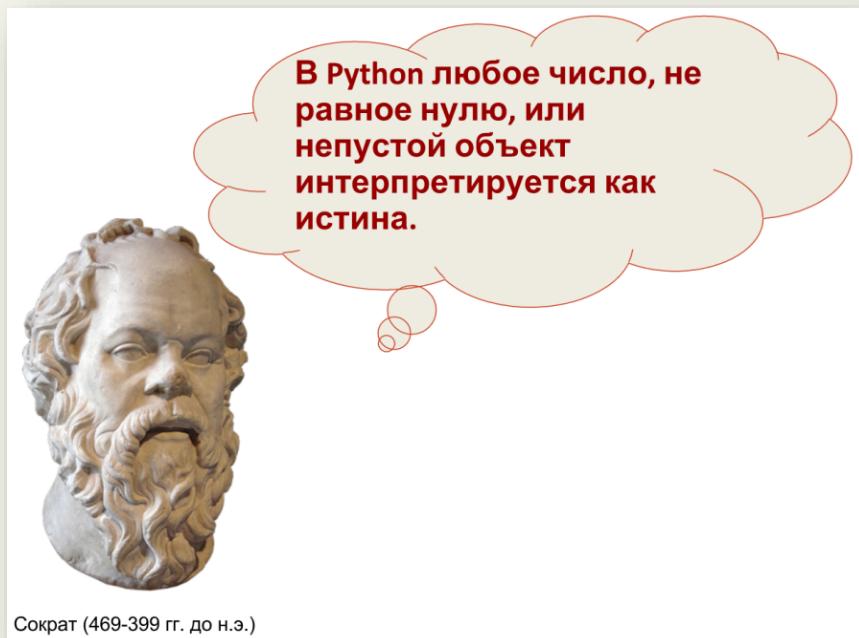
С целыми числами все более-менее просто и понятно. Рассмотрим теперь более сложные логические выражения.

Логическим высказыванием (предикатом)²¹ будем называть любое повествовательное предложение, в отношении которого можно однозначно сказать, истинно оно или ложно.

¹⁹ Важно писать в большие буквы.

²⁰ `True` интерпретируется Python как число 1, а `False` как число 0.

²¹ Подробнее здесь: http://book.kbsu.ru/theory/chapter5/1_5_1.html



В Python любое число, не равное нулю, или непустой объект интерпретируется как истина. Числа, равные нулю, пустые объекты и специальный объект `None`²³ интерпретируются как ложь.

Рассмотрим пример:

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

Мы выполнили логическую операцию `and` (И) для двух объектов: пустого строкового объекта (он будет ложным) и ненулевого числового объекта (он будет истинным). В итоге Python вернул нам пустой строковый объект. В чем тут дело?

Затем мы выполнили аналогично операцию `or` (ИЛИ). В результате получили числовой объект. Будем разбираться.

У Python есть три логических оператора `and`, `or`, `not`. `not` из них самый простой:

```
>>> y = 6 > 8  
>>> y  
False  
>>> not y  
True  
>>> not None  
True  
>>> not 2  
False
```

²³ Он имеет тип `NoneType`

Логические выражения можно комбинировать:

```
>>> 1 + 3 > 7    # приоритет + выше, чем >
False
>>> (1 + 3) > 7 # скобки способствуют наглядности
False
>>> 1 + (3 > 7)
1
>>>
```

В Python можно проверять принадлежность интервалу:

```
>>> x = 0
>>> -5 < x < 10   # эквивалентно: x > -5 and x < 10
True
>>>
```

Теперь вы без труда сможете разобраться в работе следующего кода:

```
>>> x = 5 < 10    # True
>>> y = 2 > 3     # False
>>> x or y
True
>>> (x or y) + 6
7
>>>
```

Решим небольшую задачку. Как вычислить $1/x$, чтобы не возникало ошибки деления на нуль. Для этого достаточно воспользоваться логическим оператором.

Прямой путь приводит к ошибке:

```
>>> x = 0
>>> 1 / x
Traceback (most recent call last):
  File "<pyshell#88>", line 1, in <module>
    1 / x
ZeroDivisionError: division by zero
>>>
```

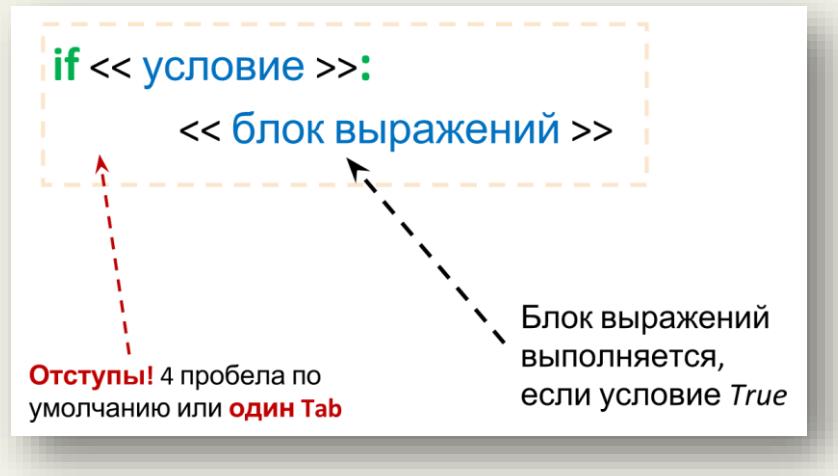
Ответом будет следующий код:

```
>>> x = 1
>>> x and 1/x
1.0
>>> x = 0
>>> x and 1/x
0
```

В качестве упражнения подумайте, почему так можно сделать.

ГЛАВА 5. УСЛОВНАЯ ИНСТРУКЦИЯ IF

Наиболее часто логические выражения используются внутри условной инструкции `if`:



Блок выражений выполняется только в том случае, если выражение, которое находится в условии, является истинным.

Для примера обратимся к таблице с водородными показателями из Википедии для различных веществ.

Произведем проверку:

```

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

```

В примере переменной `pH` присваивается вещественное значение `5.0`. Затем значение переменной сравнивается с водородным показателем для кофе и, если они совпадают, то вызывается функция `print`.

Вещество	pH
Электролит в свинцовых аккумуляторах	<1,0
Желудочный сок	1,0–2,0
Лимонный сок (5 % р-р лимонной кислоты)	2,0±0,3
Пищевой уксус	2,4
Кока-кола	3,0±0,3
Яблочный сок	3,0
Пиво	4,5
Кофе	5,0
Шампунь	5,5
Чай	5,5
Кожа здорового человека	5,5
Кислотный дождь	< 5,6
Питьевая вода	6,5–8,5
Слюна	6,8–7,4 [1]
Молоко	6,6–6,93
Чистая вода при 25 °C	7,0
Кровь	7,36–7,44
Морская вода	8,0
Мыло (жировое) для рук	9,0–10,0
Нашатырный спирт	11,5
Отбеливатель (хлорная известь)	12,5
Концентрированные растворы щелочей	>13

Рассмотрим первую большую программу (наберите ее и выполните в отдельном файле):

```
pH = float(input("Введите pH: ")) # строку преобразовали к вещественному типу

if pH == 7.0:
    print(pH, "Вода")
elif 7.36 < pH < 7.44:
    print(pH, "Кровь")
else:
    print("Что это?!")
```

Далее еще более «сложный» пример (также запустите его в отдельном файле и следите за отступами – в Python это чрезвычайно важно):

```
value = input("Введите pH: ")
if len(value) > 0:           # проверяем, что пользователь хоть что-нибудь ввел
    pH = float(value)        # переводим в вещественное число ввод пользователя
    if pH == 7.0:            # вложенный if
        print(pH, "Вода")
    elif 7.36 < pH < 7.44:
        print(pH, "Кровь")
    else:
        print("Что это?!")
else:
    print("Введите значение pH!")
```

Чтобы научиться программировать – необходимо экспериментировать: изменять код, дописывать его и смотреть, что при этом произойдет.

Для справки. Строки документации

Вспомните, когда мы вызывали функцию `help(len)`, получали справочную информацию для `len`. Откуда Python ее берет? Ответ – из самой функции.

Напишем собственную функцию, которая ничего не будет делать (в теле функции для этого указывается слово `pass`), но которая гордо объявит, что она ничего не делает.

В отдельном файле наберите и исполните:

```
def my_function():
    """Не делаем ничего, но документируем.
    Нет, правда, эта функция ничего не делает.
    """
    pass
help(my_function)
```

Результат запуска программы:

```
===== RESTART: C:/Python35-32/1.py =====
Help on function my_function in module __main__:

my_function()
    Не делаем ничего, но документируем.
    Нет, правда, эта функция ничего не делает.
>>>
```

ГЛАВА 6. МОДУЛИ В PYTHON

К примеру, вы написали несколько полезных функций, которые часто используете в своих программах. Чтобы к ним быстро обращаться, удобно все эти функции поместить в отдельный файл и загружать их оттуда. В Python такие файлы с набором функций называются модулями. Для того чтобы воспользоваться функциями, которые находятся в этом модуле, его необходимо импортировать с помощью команды `import`:

```
>>> import math  
>>>
```

Мы загрузили в память стандартный модуль `math` (содержит набор математических функций), теперь можно обращаться к функциям, находящимся внутри этого модуля. Сила Python в огромном количестве стандартных и полезных модулей.

Обратиться к функции модуля (в данном случае для нахождения квадратного корня из 9) можно следующим образом:

```
>>> math.sqrt(9)  
3.0  
>>>
```

Мы указываем имя модуля, точку и имя функции с аргументами.

Узнать о функциях, которые содержит модуль, можно через справку:

```
>>> help(math)  
Help on built-in module math:  
  
NAME  
    math  
  
DESCRIPTION  
    This module is always available. It provides access to the  
    mathematical functions defined by the C standard.  
  
FUNCTIONS  
    acos(...)  
    acos(x)  
  
        Return the arc cosine (measured in radians) of x.  
    ...  
>>>
```

Если хотим посмотреть описание конкретной функции модуля, то вызываем справку отдельно для нее:

```
>>> help(math.sqrt)  
Help on built-in function sqrt in module math:  
  
sqrt(...)  
    sqrt(x)  
  
        Return the square root of x.  
>>>
```

Мы создали собственную функцию с именем `sqrt`, затем вызвали ее и убедились, что она работает. После этого импортировали функцию `sqrt` из модуля `math`. Снова вызвали `sqrt` и видим, что это не наша функция! Ее подменили!

Теперь другой пример:

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

>>> sqrt(6)
36
>>> import math
>>> math.sqrt(9)
3.0
>>> sqrt(7)
49
>>>
```

Снова создаем собственную функцию с именем `sqrt` и вызываем ее. Затем импортируем модуль `math` и через него вызываем стандартную функцию `sqrt`. Видим, что корень квадратный считается и наша функция осталась в сохранности. Выводы сделайте самостоятельно.

В самом начале занятий мы вызывали функции для работы с числами, например, `abs` для нахождения модуля числа. На самом деле, эта функция тоже находится в модуле, который Python загружает в память в момент начала работы. Этот модуль называется `__builtins__` (два нижних подчеркивания до и после имени модуля). Если вызывать справку для данного модуля, то увидите, что там огромное количество функций и переменных:

```
>>> help(__builtins__)
Help on built-in module builtins:

NAME
    builtins - Built-in functions, exceptions, and other objects.

DESCRIPTION
    Noteworthy: None is the 'nil' object; Ellipsis represents
    '...' in slices.
...
>>>
```

В Python есть полезная функция `dir`, которая возвращает перечень имен всех функций и переменных, содержащихся в модуле:

```
>>> dir(__builtins__)
...
>>>
```

Часть из этих функций вы уже знаете, с другими – мы познакомимся чуть позже.

мы изменили наш модуль и хотим его импортировать повторно. Делается это следующим образом:

```
>>> import imp
>>> imp.reload(mtest)
test
<module 'mtest' from 'C:\\\\Python35-32\\\\mtest.py'>
>>>
```

Таким образом, мы принудительно указали Python, что модуль требует повторной загрузки. После вызова функции `reload` с указанием в качестве аргумента имени модуля, обновленный модуль загрузится повторно.

Продолжим эксперименты с модулями в Python. Создадим еще один модуль с именем `mypr.py`:

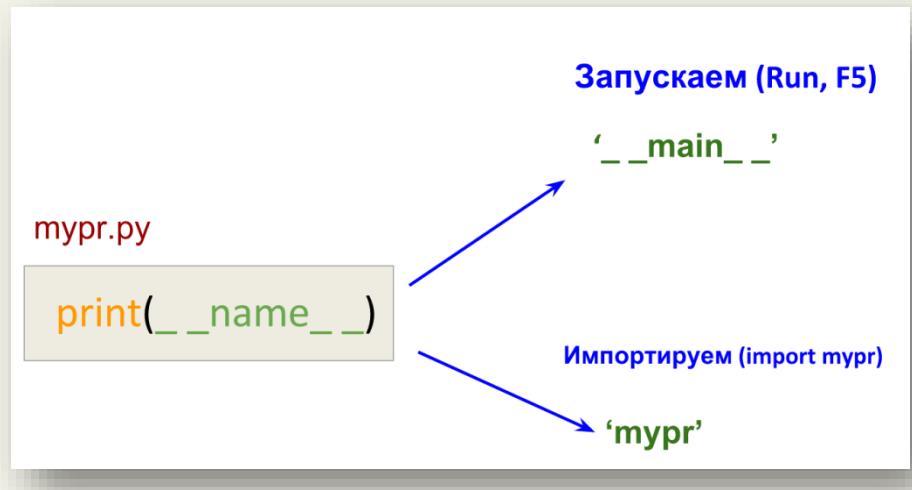
```
def func(x):
    return x**2 + 7

x = int(input("Введите значение: "))
print(func(x))
```

Импортирование модуля приводит к выполнению всей программы:

```
>>> import mypr
Введите значение: 111
12328
>>>
```

Как быть и что сделать, если мы хотим только импортировать функцию `func()` из модуля для использования ее в другой программе? Для того чтобы отделить исполнение модуля (`Run → Run Module`) от его импортирования (`import mypr`) в Python есть специальная переменная `__name__` (Python начинает названия специальных функций и переменных с двух нижних подчеркиваний):



Упражнение 7.4

Напишите программу-игру в виде отдельного модуля. Компьютер загадывает случайное число, пользователь пытается его угадать. Программа запрашивает число ОДИН раз. Если число угадано, то выводим на экран «Победа», иначе – «Повторите еще раз». Для написания программы понадобится функция `randint` из модуля `random`²⁶.

Для справки. Автоматизированное тестирование функций

В отдельном файле создайте и выполните (Run → Run Module) следующий код:

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

    >>> func_m(20, 30, 70)
    40.0

    >>> func_m(1, 5, 8)
    4.667

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

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

В результате программа отработает, но ничего не выведет на экран. Это хорошо. Разберемся, что произошло.

В программировании существует подход, при котором сначала разрабатываются тесты, т.е. как программа (функция) должна работать, а после этого пишут саму программу (функцию). Это позволяет впоследствии проверить правильность ее написания. Выше приведен пример такого подхода. Тесты помещаются в описание функции. Представим, что мы уже создали функцию `func_m`, которая вычисляет среднее арифметическое, округляя его до трех знаков после запятой, т.е. как бы мы вызвали функцию:

```
>>> func_m(20, 30, 70)
40.0
>>> func_m(1, 5, 8)
4.667
>>>
```

Мы предварительно написали проверочные тесты. Теперь мы реализуем функцию `func_m` и в ее описание добавим наши тесты. Затем импортируем модуль `doctest` и вызовем функцию `testmod`, которая запустит текущий модуль и проверит, совпадают ли результаты вызовов функций в описании с тем, что получается в реальности. Если все совпадает, то на экране ничего не появится, а если не совпадают, то отобразятся ошибки. Исправим тестовые вызовы в нашей программе:

²⁶ <https://docs.python.org/3/library/random.html>

Для справки. Философия Python

Если импортировать модуль с именем `this`, то Python отобразит на экране свою философию:

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Упражнение 7.6

Найдите значения выражений:

$$z = \frac{x + \frac{2+y}{x^2}}{\frac{1}{y + \frac{1}{\sqrt{x^2+10}}}} \text{ и } q = 2,8 \sin x + |y|$$

Упражнение 7.7

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

$$f = \begin{cases} \sin x & \text{при } 0,2 \leq x \leq 0,9, \\ 1 & \text{в противном случае.} \end{cases}$$

Упражнение 7.8

Напишите программу для моделирования бросания игрального кубика каждым из двух игроков. Определить, кто из игроков получил на кубике больше очков.

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

Метод возвращает строку, центрированную по заданной длине. По умолчанию заполняется пробелами

Аргумент задает длину строки

Форма вызова метода через обращение к его классу через точку называется *полной формой*. Постоянно писать имя класса перед вызовом каждого метода быстро надоест, поэтому чаще всего используют сокращенную форму вызова метода:

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

В примере мы вынесли первый аргумент метода и поместили его вместо имени класса:

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

Вынесли из аргумента (могут быть выражением)

В момент, когда мы используем сокращенную форму для вызова метода, Python преобразует ее в полную форму, а затем вызывает. Это знание нам пригодится, когда дойдем до изучения ООП.

Для вызова справки у методов необходимо через точку указывать их класс:

```
>>> help(str.capitalize)
Help on method_descriptor:

capitalize(...)
    S.capitalize() -> str
```

Return a capitalized version of S, i.e. make the first character have upper case and the rest lower case.

```
>>>
```

В примере Python вернул строку, очищенную от символа переноса строки (\n) и пробелов.

Метод swapcase возвращает строку с противоположными регистрами символов:

```
>>> 'Hello'.swapcase()
'hELLO'
>>>
```

Python позволяет творить чудеса с вызовами методов – их можно вызывать подряд в одну строку:

```
>>> 'ПРИВЕТ'.swapcase().endswith('т')
True
>>>
```

В первую очередь вызывается метод swapcase для строки 'ПРИВЕТ', затем для результирующей строки вызывается метод endswith с аргументом 'т':

"ПРИВЕТ".swapcase().endswith('Т')

'привет'.endswith('Т')

True

Рассмотрим перечень популярных строковых методов.

Рекомендую каждый из перечисленных ниже методов запустить в интерактивном режиме на примере различных строк.

Предположим, что переменная s содержит некоторую строку, тогда применим к ней методы²⁸:

s.upper() – возвращает строку в верхнем регистре
s.lower() – возвращает строку в нижнем регистре
s.title() – возвращает строку, первый символ которой в верхнем регистре
s.find('вет', 2, 3) – возвращает позицию подстроки в интервале либо -1
s.count('е', 1, 5) – возвращает количество подстрок в интервале либо -1
s.isalpha() – проверяет, состоит ли строка только из букв
s.isdigit() – проверяет, состоит ли строка только из чисел
s.isupper() – проверяет, написаны ли все символы в верхнем регистре
s.islower() – проверяет, написаны ли все символы в нижнем регистре
s.istitle() – проверяет, начинается ли строка с большой буквы
s.isspace() – проверяет, состоит ли строка только из пробелов

²⁸ Документация: <https://docs.python.org/3/library/stdtypes.html#string-methods>

ГЛАВА 9. СПИСКИ В PYTHON

9.1. Создание списка

Начнем с примера. Предположим, что нам необходимо обработать информацию о курсах валют²⁹:

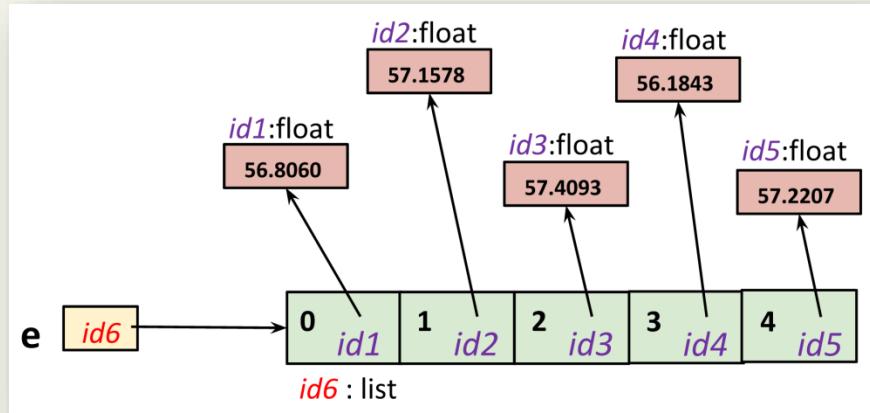
Дата	Доллар США USD	ЕВРО EUR
16.05.2015	50.0115	56.9881
15.05.2015	50.0774	57.1383
14.05.2015	49.5366	55.7138
13.05.2015	50.9140	57.1102
09.05.2015	50.7511	56.8971
08.05.2015	50.3615	57.2207
07.05.2015	49.9816	56.1843
06.05.2015	51.7574	57.4093
01.05.2015	51.1388	57.1578
30.04.2015	51.7029	56.8060
29.04.2015	52.3041	56.9016
28.04.2015	51.4690	55.8747
25.04.2015	50.2473	54.6590
24.04.2015	51.6011	55.1255
23.04.2015	53.6555	57.7226
22.04.2015	53.9728	57.5998

Мы можем курс валюты на каждый день поместить в отдельную переменную:

```
>>> day1 = 56.8060
>>> day2 = 57.1578
>>>
```

²⁹ <http://www.sberometer.ru/cbr/>

Рассмотрим, как Python работает со списками в памяти:



Видим, что переменная `e` содержит адрес списка (`id6`). Каждый элемент списка является указателем (хранит адрес) другого объекта (в данном случае вещественных чисел).

В общем виде создание списка выглядит следующим образом:

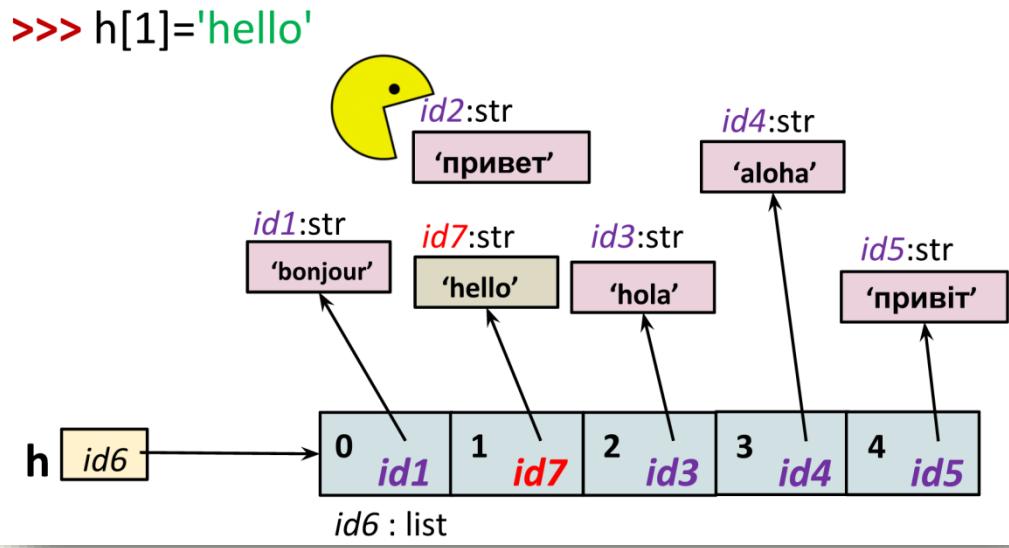


Отмечу, что на месте элементов списка могут находиться выражения, а не просто отдельные объекты.

9.2. Операции над списками

Обращаться к отдельным элементам списка можно по их индексу (позиции), начиная с нуля:

```
>>> e = [56.8060, 57.1578, 57.4093, 56.1843, 57.2207]
>>> e[0]
56.806
>>> e[1]
57.1578
>>> e[-1]      # последний элемент
57.2207
>>>
```



В момент изменения списка в памяти создается новый строковый объект 'hello'. Затем адрес на этот объект (*id7*) помещается в первую ячейку списка (вместо *id2*). Python увидит, что на объект по адресу *id2* нет ссылок, поэтому удалит его из памяти.

Список (list), наверное, наиболее часто встречающийся тип данных, с которым приходится сталкиваться при написании программ. Это связано со встроенными в Python функциями, которые позволяют легко и быстро обрабатывать списки:

len(L) – возвращает число элементов в списке L
max(L) – возвращает максимальное значение в списке L
min(L) – возвращает минимальное значение в списке L
sum(L) – возвращает сумму значений в списке L
sorted(L) – возвращает копию списка L, в котором элементы упорядочены по возрастанию. Не изменяет список L

Примеры вызовов функций:

```

>>> e = [56.8060, 57.1578, 57.4093, 56.1843, 57.2207]
>>> e
[56.806, 57.1578, 57.4093, 56.1843, 57.2207]
>>> len(e)
5
>>> max(e)
57.4093
>>> min(e)
56.1843
>>> sum(e)
284.7781
>>> sorted(e)
[56.1843, 56.806, 57.1578, 57.2207, 57.4093]
>>> e
[56.806, 57.1578, 57.4093, 56.1843, 57.2207]
>>>
  
```

Получилась небольшая функция, которая может объединять и складывать в зависимости от класса (типа данных) переданных ей объектов.

Следующий полезный оператор `in`³¹ (схожим образом работает для строк):

```
>>> h = ['bonjour', 7, 'hola', -1.0, 'привіт']
>>> if 7 in h:
    print('Значение есть в списке')
```

```
Значение есть в списке
>>>
```

Упражнение 9.2

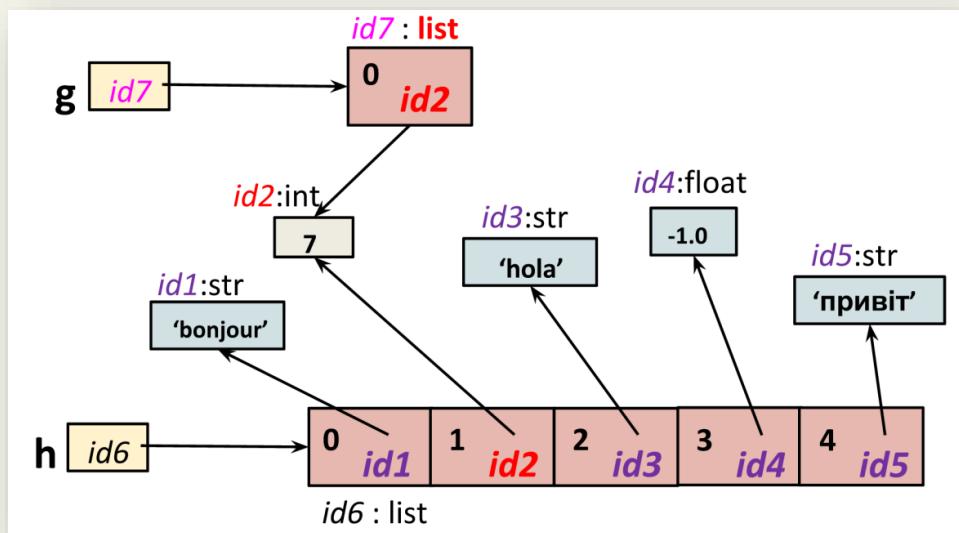
```
L = [3, 'hello', 7, 4, 'привет', 4, 3, -1]
```

Определите наличие строки «привет» в списке. ЕСЛИ такая строка в списке присутствует, ТО вывести ее на экран, повторив 10 раз.

Аналогично строкам для списка есть операция взятия среза:

```
>>> h = ['bonjour', 7, 'hola', -1.0, 'привіт']
>>> h
['bonjour', 7, 'hola', -1.0, 'привіт']
>>> g = h[1:2]
>>> g
[7]
>>>
```

В памяти это выглядит следующим образом:



Переменной `g` присваивается адрес нового списка (`id7`), содержащего указатель на числовой объект, выбранный с помощью среза.

³¹ Для списков `in` выполняет линейный поиск, что сильно сказывается на скорости работы для очень больших наборов данных.

Создание псевдонимов – особенность списков, т.к. они могут изменяться. Будьте крайне внимательны.

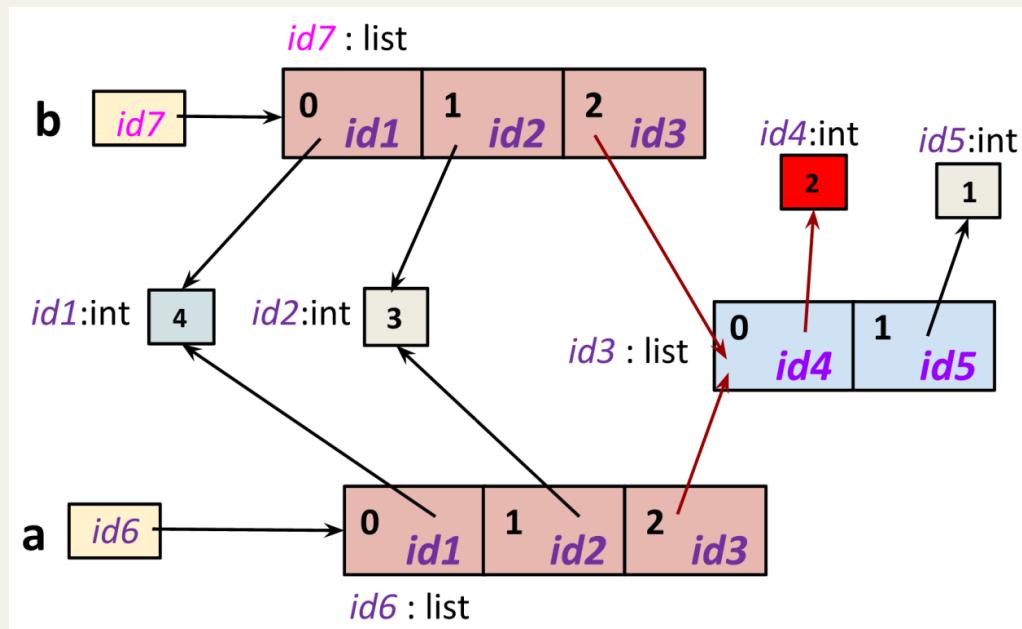
Возникает вопрос, как проверить, ссылаются ли переменные на один и тот же список:

```
>>> x = y = [1, 2] # создали псевдонимы
>>> x is y # проверка, ссылаются ли переменные на один и тот же объект33
True
>>> x = [1, 2]
>>> y = [1, 2]
>>> x is y
False
>>>
```

К спискам применимы два вида копирования. Первый вид – *поверхностное копирование*, при котором создается новый объект, но он будет заполнен ссылками на элементы, которые содержались в оригинале:

```
>>> a = [4, 3, [2, 1]]
>>> b = a[:]
>>> b is a
False
>>> b[2][0] = -100
>>> a
[4, 3, [-100, 1]] # список a тоже изменился
>>>
```

Следующий рисунок демонстрирует схему размещения ссылок на объекты при поверхностном копировании:



³³ С помощью `is` сравниваются ссылки-адреса, а не сами объекты.

Еще несколько полезных методов для списка:

```
>>> colors
['red', 'orange', 'yellow', 'green', 'blue', 'purple']
>>> colors.pop() # удаляет и возвращает последний элемент списка
'purple'
>>> colors
['red', 'orange', 'yellow', 'green', 'blue']
>>> colors.reverse() # список в обратном порядке
>>> colors
['blue', 'green', 'yellow', 'orange', 'red']
>>> colors.sort() # сортирует список (вспомните о сравнении строк)
>>> colors
['blue', 'green', 'orange', 'red', 'yellow']
>>> colors.clear() # очищает список. Метод появился в версии 3.3. Аналог del color[:]
>>> colors
[]
>>>
```

Методов много, поэтому для их запоминания рекомендую выполнить каждый из перечисленных выше методов для различных аргументов и посмотреть, что они возвращают. Это обязательно пригодится при написании программ.

9.5. Преобразование типов

Очень часто появляется потребность в изменении строк, но напрямую мы этого сделать не можем. Тогда нам на помощь приходят списки. Преобразуем строку в список, изменим список, затем вернем его в строку:

```
>>> s = 'Строка для изменения'
>>> list(s)    # функция list() пытается преобразовать аргумент в список
['С', 'т', 'р', 'о', 'к', 'а', ' ', 'д', 'л', 'я', ' ', 'и', 'з',
'м', 'е', 'н', 'е', 'н', 'и', 'я']
>>> lst = list(s)
>>> lst[0] = 'М' # изменяем список, полученный из строки
>>> lst
['М', 'т', 'р', 'о', 'к', 'а', ' ', 'д', 'л', 'я', ' ', 'и', 'з',
'м', 'е', 'н', 'е', 'н', 'и', 'я']
>>> s = ''.join(lst) # преобразуем список в строку с помощью строкового метода join()
>>> s
'Мтрока для изменения'
>>>
```

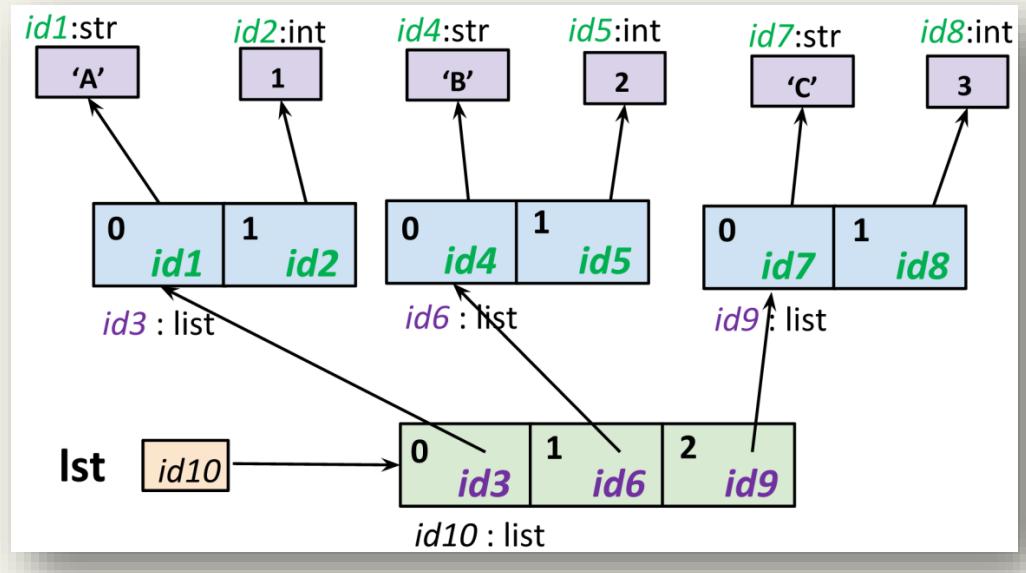
Отдельно рассмотрим несколько примеров строкового метода `join()`:

```
>>> A = ['red', 'green', 'blue']
>>> ''.join(A)
'red green blue'
>>> ''.join(A)
'redgreenblue'
>>> '***'.join(A)
'red***green***blue'
>>>
```

Обращение (изменение) к вложенному списку происходит через указание двух индексов:

```
>>> lst[0][1]  
1  
>>>
```

Схематично вложенные списки выглядят следующим образом:



Упражнение 9.5

Задан список слов. Необходимо выбрать из него случайное слово. Из выбранного случайного слова случайно выбрать букву и попросить пользователя ее угадать.

```
Задан список слов: ['самовар', 'весна', 'лето']  
Выбираем случайное слово: 'весна'  
Выбираем случайную букву: 'с'  
Выводим на экран: ве?на  
Пользователь пытается угадать букву.
```

Подсказка: используйте метод `choice` модуля `random`.

В общем виде цикл `for` для перебора всех элементов указанного списка выглядит следующим образом:

```
for << переменная >> in << список >>:  
    << тело цикла >>
```

Небольшой пример:

```
>>> for i in [1, 2, 'hi']:  
        print(i)  
  
1  
2  
hi  
>>>
```

На самом деле, цикл `for` работает и для строк!

```
>>> for i in 'hello':  
        print(i)  
  
h  
e  
l  
l  
o  
>>>
```

По аналогии со списком для строк перебираются все символы строки.

В общем виде запись цикла `for` для заданной строки:

```
for << переменная >> in << строка >>:  
    << тело цикла >>
```

Цикл `for` позволяет не только выводить элементы строки или списка на экран, но и производить над ними определенные операции:

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

Дело в том, что для создания диапазона чисел необходимо использовать цикл `for`:

```
>>> for i in range(0, 10, 1):
    print(i, end=' ')
0 1 2 3 4 5 6 7 8 9
>>> for i in range(10):
    print(i, end=' ')
0 1 2 3 4 5 6 7 8 9
>>> for i in range(2, 20, 2):
    print(i, end=' ')
2 4 6 8 10 12 14 16 18
>>>
```

Таким образом, в переменную `i` на каждом шаге цикла будет записываться значение из диапазона, который создается функцией `range`.

При желании можно получить диапазон в обратном порядке следования (обратите внимание на аргументы функции `range`):

```
>>> for i in range(20, 2, -2):
    print(i, end=' ')
20 18 16 14 12 10 8 6 4
>>>
```

Теперь с помощью диапазона найдем сумму чисел на интервале от 1 до 100:

```
>>> total = 0
>>> for i in range(1, 101):
    total = total + i # total += i

>>> total
5050
>>>
```

Переменной `i` на каждом шаге цикла будет присваиваться значение из диапазона от 1 до 100 (крайнее значение не включаем). В цикле мы накапливаем счетчик. Что это означает? На первом шаге цикла сначала вычисляется правая часть выражения, т.е. `total+i`. Переменная `total` на первом шаге равна 0 (присвоили ей значение 0 перед началом цикла), переменная `i` на первом шаге содержит значение 1 (первое значение из диапазона), таким образом, правая часть будет равна значению 1 и это значение присвоится левой части выражения, т.е. переменной `total`.

На втором шаге `total` уже будет равна значению 1, `i` – содержать значение 2, т.е. правая часть выражения будет равна 3, это значение присвоится снова `total` и т.д. пока не дойдем до конца диапазона. В итоге в `total` после выхода из цикла будет содержаться искомая сумма!

Рассмотрим различные способы создания списков. Самый очевидный способ:

```
>>> a = []
>>> for i in range(1,15):
    a.append(i)
>>> a
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>>
```

В цикле из диапазона от 1 до 14 выбираем числа и с помощью спискового метода append добавляем их к списку a.

С созданием списка из диапазона мы уже встречались:

```
>>> a = list(range(1, 15))
>>> a
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>>
```

Можно также использовать «списковое включение» (иногда его называют «генератором списка»):

```
>>> a = [i for i in range(1, 15)]
>>> a
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>>
```

Правила работы для спискового включения:



В следующем примере выбираем из диапазона все числа от 1 до 14, возводим их в квадрат и сразу формируем из них новый список:

```
>>> a = [i**2 for i in range(1,15)]
>>> a
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196]
>>>
```

Списковое включение позволяет задавать условие для выбора значения из диапазона (в примере исключили значение 4):

```
>>> a = [i**2 for i in range(1,15) if i!=4]
>>> a
[1, 4, 9, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196]
>>>
```

Вместо диапазонов списковое включение позволяет указывать существующий список:

`randint`, которая сгенерирует целое случайное число из интервала, и уже это число добавится в новый список.

Перейдем к ручному вводу значений для списка. Зададим длину списка и введем с клавиатуры все его значения:

```
a = [] # объявляем пустой список
n = int(input()) # считываем количество элементов в списке
for i in range(n):
    new_element = int(input()) # считываем очередной элемент
    a.append(new_element) # добавляем его в список
    # последние две строки можно было заменить одной:
    # a.append(int(input()))
print(a)
```

В результате запуска программы:

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

В этом примере `range` снова выступает как счетчик числа повторений, а именно – задает длину списка.

Теперь запишем решение этой задачи через списковое включение в одну строку:

```
>>> A = [int(input()) for i in range(int(input()))]
3
4
2
1
>>> A
[4, 2, 1]
>>>
```

Упражнение 10.3

Дан список числовых значений, насчитывающий N элементов. Поменяйте местами первую и вторую половины списка.

Рассмотрим следующий пример:

```
while True:
    text = input("Введите число или стоп для выхода: ")
    if text == "стоп":
        print("Выход из программы! До встречи!")
        break      # инструкция выхода из цикла
    elif text == '1':
        print("Число 1")
    else:
        print("Что это?!")
```

В результате работы программы получим:

```
>>>
=====
RESTART: C:\Python35-32\myprog.py =====
Введите число или стоп для выхода: 4
Что это?!
Введите число или стоп для выхода: 1
Число 1
Введите число или стоп для выхода: стоп
Выход из программы! До встречи!
>>>
```

Программа выполняется в бесконечном цикле, т.к. `True` всегда является истиной. Внутри цикла происходит ввод значения с клавиатуры и проверка введенного значения. Инструкция `break` осуществляет выход из цикла.

В подобных программах необходимо внимательно следить за преобразованием типов данных.

Упражнение 10.4

Напишите программу-игру. Компьютер загадывает случайное число, пользователь пытается его угадать. Пользователь вводит число до тех пор, пока не угадает или не введет слово «Выход». Компьютер сравнивает число с введенным и сообщает пользователю больше оно или меньше загаданного.

В следующей программе реализован один из вариантов подсчета суммы чисел в строке:

```
s = 'aa3aBbb6ccc'
total = 0
for i in range(len(s)):
    if s[i].isalpha():    # посимвольно проверяем наличие буквы
        continue          # инструкция перехода к следующему шагу цикла
    total=total+int(s[i]) #накапливаем сумму, если встретилась цифра

print("сумма чисел:", total)
```

10.5. Вложенные циклы

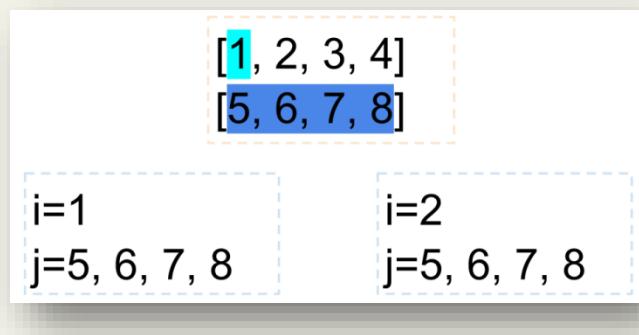
Циклы можно вкладывать друг в друга.

```
outer = [1, 2, 3, 4]      # внешний цикл
inner = [5, 6, 7, 8]      # вложенный (внутренний) цикл
for i in outer:
    for j in inner:
        print('i=', i, 'j=', j)
```

Результат работы программы:

```
>>>
=====
RESTART: C:\Python35-32\myprog.py =====
i= 1 j= 5
i= 1 j= 6
i= 1 j= 7
i= 1 j= 8
i= 2 j= 5
i= 2 j= 6
i= 2 j= 7
i= 2 j= 8
i= 3 j= 5
i= 3 j= 6
i= 3 j= 7
i= 3 j= 8
i= 4 j= 5
i= 4 j= 6
i= 4 j= 7
i= 4 j= 8
>>>
```

В примере цикл `for` сначала продвигается по всем элементам внешнего цикла (фиксируем `i=1`), затем переходит к вложенному циклу (переменная `j`) и проходим по всем элементам вложенного списка. Далее возвращаемся к внешнему циклу (фиксируем следующее значение `i=2`) и снова проходим по всем элементам вложенного списка. Так повторяем до тех пор, пока не закончатся элементы во внешнем списке:



Данный прием активно используется при работе с вложенными списками.

Для справки. Else в инструкции цикла

Инструкции циклов могут иметь ветвь `else`. Она исполняется, когда цикл выполнил перебор до конца (в случае `for`) или когда условие становится ложным (в случае `while`), но не в тех случаях, когда цикл прерывается по `break`.

Рассмотрим следующий пример разложения числа на множители:

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'равно', x, '*', n//x)
            break
    else:
        # циклу не удалось найти множитель
        print(n, '- простое число')
```

Результат выполнения программы:

```
>>>
=====
RESTART: C:\Python35-32\myprog.py =====
2 - простое число
3 - простое число
4 равно 2 * 2
5 - простое число
6 равно 2 * 3
7 - простое число
8 равно 2 * 4
9 равно 3 * 3
>>>
```

```
[3, 5, 6]
>>>
```

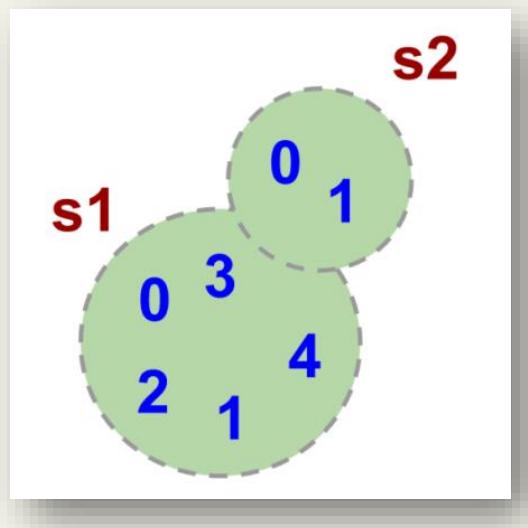
Функция `range` позволяет создавать множества из диапазона:

```
>>> set(range(10))
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>>
```

Рассмотрим некоторые операции над множествами³⁷:

```
>>> s1 = set(range(5))
>>> s2 = set(range(2))
>>> s1
{0, 1, 2, 3, 4}
>>> s2
{0, 1}
>>> s1.add('5')    # добавить элемент
>>> s1
{0, 1, 2, 3, 4, '5'}
>>>
```

У множеств в Python много общего с множествами из математики:



```
>>> s1.intersection(s2) # пересечение множеств через вызов метода (s1 & s2)
{0, 1}
>>> s1.union(s2)       # объединение множеств через вызов метода (s1 | s2)
{0, 1, 2, 3, 4, '5'}
>>>
```

³⁷ <https://docs.python.org/3/tutorial/datastructures.html#sets>

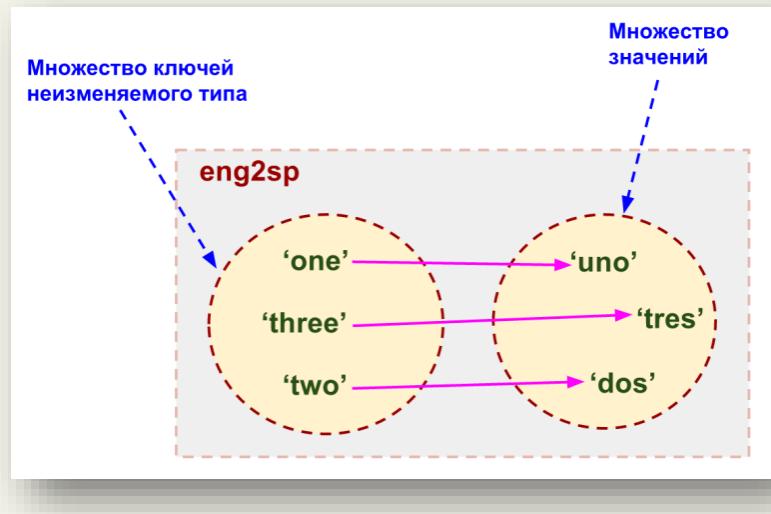
Мы сказали, что кортеж нельзя изменить, но можно изменить, например, список, входящий в кортеж:

```
>>> t = (1, [1, 3], '3')
>>> t[1]
[1, 3]
>>> t[1][0] = '1'
>>> t
(1, ['1', 3], '3')
>>>
```

11.3. Словари

Следующий тип данных (класс) – словарь (dict). Словарь в Python – неупорядоченная изменяемая коллекция или, проще говоря, «список» с произвольными ключами, неизменяемого типа.

Пример создания словаря, который каждому слову на английском языке будет ставить в соответствие слово на испанском языке.



```
>>> eng2sp = dict()      # создаем пустой словарь
>>> eng2sp
{}
>>> eng2sp['one'] = 'uno' # добавляем 'uno' для элемента с индексом 'one'
>>> eng2sp
{'one': 'uno'}
>>> eng2sp['one']
'uno'
>>> eng2sp['two'] = 'dos'
>>> eng2sp['three'] = 'tres'
>>> eng2sp
{'three': 'tres', 'one': 'uno', 'two': 'dos'}
>>>
```

Для справки. Переменное число параметров

Когда мы объявляем параметр со звездочкой (например, `*param`), все **позиционные аргументы**, начиная с этой позиции и до конца, будут собраны в кортеж под именем `param`. Аналогично, когда мы объявляем параметры с двумя звездочками (`**param`), все **ключевые аргументы**, начиная с этой позиции и до конца, будут собраны в словарь под именем `param`.

```
def total(initial=5, *numbers, **keywords):
    count = initial
    for number in numbers:
        count += number      # или count = count + number
    for key in keywords:
        count += keywords[key]  # или count = count + keywords[key]
    return count

# 1, 2, 3 - позиционные аргументы, vegetables и fruits - ключевые аргументы
print(total(10, 1, 2, 3, vegetables=50, fruits=100))
```

Результат работы программы:

```
>>>
=====
RESTART: C:/Python35-32/test.py =====
166
>>>
```

Если некоторые ключевые параметры должны быть доступны только по ключу, а не как позиционные аргументы, их можно объявить после параметра со звездочкой. Объявление параметров после параметра со звездочкой дает только ключевые аргументы. Если для таких аргументов не указано значение по умолчанию, и оно не передано при вызове, обращение к функции вызовет ошибку

```
def total(initial=5, *numbers, extra_number):
    count = initial
    for number in numbers:
        count += number
    count += extra_number
    print(count)

total(10, 1, 2, 3, extra_number=50)
total(10, 1, 2, 3)
# Вызовет ошибку, поскольку мы не указали значение
# аргумента по умолчанию для 'extra_number'
```

Результат работы программы:

```
>>>
=====
RESTART: C:/Python35-32/test.py =====
66
Traceback (most recent call last):
  File "C:/Python35-32/test.py", line 9, in <module>
    total(10, 1, 2, 3)
TypeError: total() missing 1 required keyword-only argument: 'extra_number'
>>>
```

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

Возвращаем удаленный (первый минимальный) элемент обратно в список:

[809, 834, 477, 478, 307, 122, 96, 102, 324, 476]
индекс: 0 1 2 3 4 5 6 7 8 9

Не забываем о смещении индексов после удаления первого минимального элемента: индекс второго минимального элемента равен индексу первого минимального элемента, поэтому увеличим индекс второго минимального на 1.

Функция, реализующая данный алгоритм имеет вид:

```
def find_two_smallest(L):
    smallest = min(L)
    min1 = L.index(smallest)
    L.remove(smallest)      # удаляем первый минимальный элемент

    next_smallest = min(L)
    min2 = L.index(next_smallest)
    L.insert(min1, smallest) # возвращаем первый минимальный обратно
    if min1 <= min2: # проверяем индекс второго минимального из-за смещения
        min2 += 1      # min2 = min2 + 1

    return (min1, min2) # возвращаем кортеж
```

2. Сортировка, поиск минимальных, определение индексов

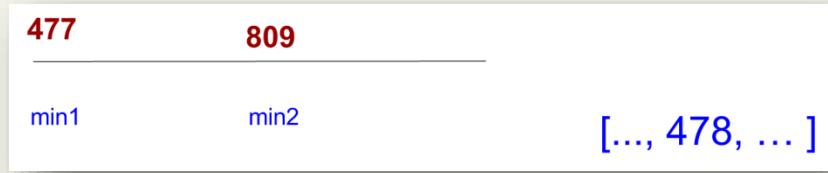
Реализация второго алгоритма интуитивно понятна, поэтому приведу только исходный текст функции:

```
def find_two_smallest(L):
    temp_list = sorted(L) # возвращаем КОПИЮ отсортированного списка
    smallest = temp_list[0]
    next_smallest = temp_list[1]
    min1 = L.index(smallest)
    min2 = L.index(next_smallest)
    return (min1, min2)
```

3. Перебор всего списка: сравниваем каждый элемент по порядку, получаем два наименьших значения, обновляем значения, если найдены наименьшие.

Третий алгоритм наиболее сложный из перечисленных выше, поэтому остановимся на нем подробнее.

Рассматриваем следующий элемент списка (478):



Он оказался между двумя минимальными элементами (условно назовем это «вторым вариантом»):



Снова обновляем минимальные элементы (теперь обновился только min2):



И т.д. пока не дойдем до конца списка:



Исходный текст функции, реализующий предложенный алгоритм:

```
def find_two_smallest(L):
    if L[0] < L[1]:
        min1, min2 = 0, 1 # устанавливаем начальные значения
    else:
        min1, min2 = 1, 0

    for i in range(2, len(L)):
        if L[i] < L[min1]: # «первый вариант»
            min2 = min1
            min1 = i
        elif L[i] < L[min2]: # «второй вариант»
            min2 = i
    return (min1, min2)
```

ГЛАВА 13. ОБРАБОТКА ИСКЛЮЧЕНИЙ В PYTHON

В этой главе речь пойдет об обработке ошибок (исключений) в Python. Рассмотрим пример:

```
x = int(input())
print(5/x)
```

Выполним его и убедимся, что перевод буквы в число и деление на нуль приводят к ошибкам:

```
>>>
=====
RESTART: C:\Python35-32\test.py =====
r
Traceback (most recent call last):
  File "C:\Python35-32\test.py", line 1, in <module>
    x = int(input())
ValueError: invalid literal for int() with base 10: 'r'
>>>
=====
RESTART: C:\Python35-32\test.py =====
0
Traceback (most recent call last):
  File "C:\Python35-32\test.py", line 2, in <module>
    print(5/x)
ZeroDivisionError: division by zero
>>>
```

Возникают два извечных вопроса: кто виноват и что делать?

Можно осуществить прямую проверку вводимого с клавиатуры значения:

```
x = int(input())
if x==0:
    print("Error!")
else:
    print(5/x)
```

Программа работает и на нуль не делит:

```
>>>
=====
RESTART: C:\Python35-32\test.py =====
0
Error!
>>>
```

Python предлагает⁴⁰ другой способ, основанный на перехвате ошибок (исключений).

⁴⁰ Другие объектно-ориентированные языки тоже поддерживают данный механизм

Воспользуемся этим знанием и перепишем нашу программу:

```
try:  
    x = int(input("Enter number: "))  
    print(5/x)  
except ZeroDivisionError: # указываем тип исключения  
    print("Error dividing by zero")  
except ValueError:  
    print("Error converting to a number")
```

Выполним программу и убедимся, что теперь срабатывают различные блоки `except` в зависимости от типа возникающих ошибок (исключений):

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

Инструкция обработки исключений имеет несколько дополнительных возможностей. Рассмотрим их на примере:

```
try:  
    x = int(input("Введите число: "))  
    print(5/x)  
except ZeroDivisionError as z:  
    print("Обрабатываем исключение - деление на нуль!")  
    print(z) # выводим на экран информацию об исключении ZeroDivisionError  
except ValueError as v:  
    print("Обрабатываем исключение - преобразование типов!")  
    print(v)  
else:  
    print("Выполняется, если не произошло исключительных ситуаций!")  
finally:  
    print("Выполняется всегда и в последнюю очередь!")
```

Запустим программу для различных входных значений:

```
>>>  
===== RESTART: C:\Python35-32\test.py =====  
Введите число: 0  
Обрабатываем исключение - деление на нуль!  
division by zero  
Выполняется всегда и в последнюю очередь!  
>>>  
===== RESTART: C:\Python35-32\test.py =====  
Введите число: r  
Обрабатываем исключение - преобразование типов!  
invalid literal for int() with base 10: 'r'  
Выполняется всегда и в последнюю очередь!  
>>>
```

ГЛАВА 14. РАБОТА С ФАЙЛАМИ В PYTHON

Чаще всего данные для обработки поступают из внешних источников – файлов. Существуют различные форматы файлов, наиболее простой и универсальный – текстовый. Он открывается в любом текстовом редакторе (например, Блокноте). Расширения у текстовых файлов: .txt, .html, .csv (их достаточно много).

Помимо текстовых есть другие типы файлов (музыкальные, видео, .doc, .ppt и пр.), которые открываются в специальных программах (музыкальный или видео проигрыватель, MS Word и пр.).

В этой главе остановимся на текстовых файлах, хотя возможности Python этим не ограничиваются.

Выполните несколько шагов.

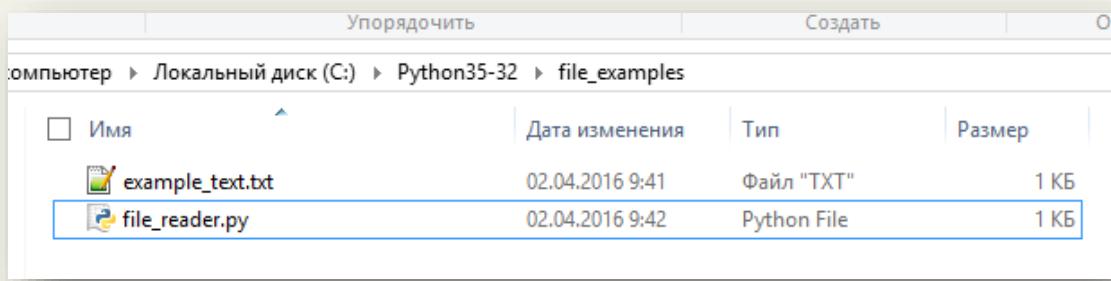
1. Создайте каталог (папку) *file_examples*.
2. С помощью (например, Блокнота) создайте в каталоге *file_examples* текстовый файл *example_text.txt*, содержащий следующий текст:

```
First line of text  
Second line of text  
Third line of text
```

3. Создайте в каталоге *file_examples* файл *file_reader.py*, содержащий исходный текст программы на языке Python:

```
file = open('example_text.txt', 'r')  
contents = file.read()  
print(contents)  
file.close()
```

Получится примерно следующее:



Запустим программу *file_reader.py*:

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

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

Исходный текст заметно упростился, т.к. освобождение ресурсов в этом случае происходит автоматически (внутри менеджера контекста).

Каким образом Python определяет, где искать файл для открытия? В момент вызова функции `open` Python ищет указанный файл в текущем *рабочем каталоге*. В момент запуска программы текущий рабочий каталог там, где сохранена программа.

Определить текущий *рабочий каталог* можно следующим образом:

```
>>> import os
>>> os.getcwd()
'C:\Python35-32\file_examples'
>>>
```

Если файл находится в другом каталоге, то необходимо указать путь к нему:

1. абсолютный путь (начиная с корневого каталога):

'C:\\Users\\Dmitriy\\data1.txt'

2. относительный путь (относительно текущего рабочего каталога, см. рисунок ниже):

'data\\data1.txt'

На следующем рисунке текущий рабочий каталог '`C:\\Users\\Dmitriy\\home`'



Далее рассмотрим некоторые способы чтения содержимого файла.

В следующем примере происходит чтение содержимого всего файла, начиная с текущей позиции курсора (перемещает курсор в конец файла):

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

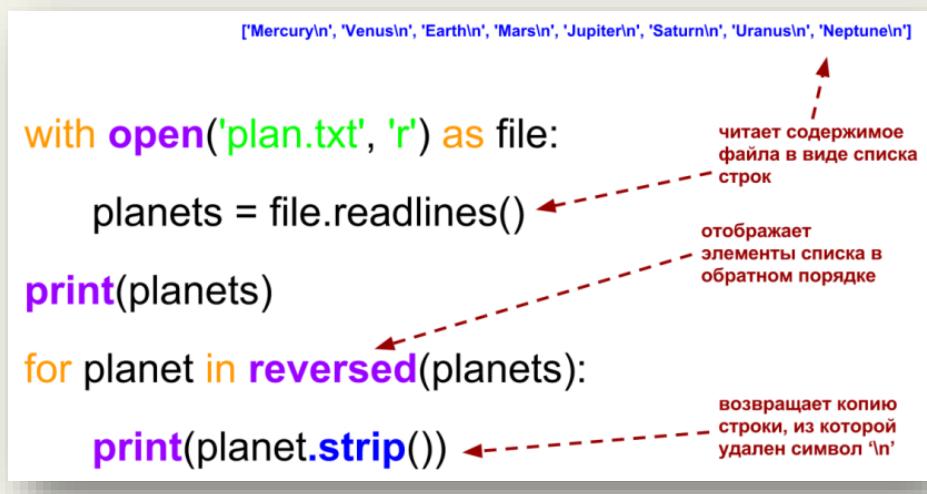
Далее запустим программу (с учетом текущего рабочего каталога!):

```
with open('plan.txt', 'r') as file:  
    planets = file.readlines()  
  
print(planets)  
  
for planet in reversed(planets):  
    print(planet.strip())
```

В результате выполнения получим:

```
>>>  
==== RESTART: C:\Python35-32\file_examples\file_reader.py ====  
['Mercury\n', 'Venus\n', 'Earth\n', 'Mars\n', 'Jupiter\n',  
 'Saturn\n', 'Uranus\n', 'Neptune']  
Neptune  
Uranus  
Saturn  
Jupiter  
Mars  
Earth  
Venus  
Mercury  
>>>
```

Комментарии к исходному тексту приведены на следующем рисунке:



Используйте следующий способ чтения из файла, если хотите сделать некоторые операции с каждой из строк, начиная с текущей позиции файлового курсора до конца файла:

```
with open('plan.txt', 'r') as file:  
    for line in file:  
        print(line)  
        print(len(line.strip()))
```

Следующий пример показывает, как можно напрямую обращаться к файлам, находящимся в сети Интернет:

```
import urllib.request
url = "http://dfedorov.spb.ru/python3/src/romeo.txt"
with urllib.request.urlopen(url) as webpage:
    for line in webpage:
        line = line.strip()
        line = line.decode('utf-8') # преобразуем тип bytes в utf-8
        print(line)
```

Для справки. Регулярные выражения

Python поддерживает мощный язык регулярных выражений, т.е. шаблоны, по которым можно искать/заменять некоторый текст.

Например, регулярное выражение '[ea]' означает любой символ из набора в скобках, т.е. регулярное выражение 'r[ea]d' совпадает с 'red' и 'radar', но не со словом 'read'.

Для работы с регулярными выражениями необходимо импортировать модуль re:

```
>>> import re
>>> re.search("r[ea]d", "rad") # указываем шаблон и текст
<_sre.SRE_Match object; span=(0, 3), match='rad'>
>>> re.search("r[ea]d", "read")
>>> re.search("[1-8]", "3") # ищет совпадением с любым числом из интервала
<_sre.SRE_Match object; span=(0, 1), match='3'>
>>> re.search("[1-8]", "9")
>>>
```

В случае совпадения текста с шаблоном возвращается объект match, иначе возвращается None.

Упражнение 14.2

Найдите в файле (файл находится в сети Интернет): <http://dfedorov.spb.ru/python/files/mbox-short.txt> строки, содержащие почтовые адреса. Запишите найденные строки в файл с именем mail.txt.

Упражнение 14.3

Очистите файл от HTML-тегов: <http://dfedorov.spb.ru/python/files/p.html>
Выполните на экран «чистый» текст. PS. можно использовать только стандартные модули Python.

Упражнение 14.4

Определите частоту встречаемости всех слов для текста, находящегося в сети Интернет: <http://dfedorov.spb.ru/python3/src/romeo.txt> PS: используйте словари (dict).

Создание и сортировка словаря по значению:

```
>>> d = {"t1":2, "t6":5, "t9":1}
>>> d
{'t9': 1, 't6': 5, 't1': 2}
>>> sorted(d)
['t1', 't6', 't9']
>>> d.get('t1')
2
>>> sorted(d, key=d.get)
['t9', 't1', 't6']
```

Упражнение 14.6*

Напишите функцию `stringCount`, которая принимает два входных аргумента – имя файла и строку, а возвращает число повторений указанной строки в указанном файле.

Упражнение 14.7*

Реализуйте функцию `myGrep`, которая принимает два входных аргумента – имя файла и строку, а выводит на экран все строки указанного файла, содержащие заданную строку в качестве подстроки:

```
>>> myGrep('example.txt', 'line')
The 3 lines in this file end with the new line character.
There is a blank line above this line.
```

Упражнение 14.8*

Реализуйте функцию `links`, которая принимает на вход имя HTML-файла и возвращает количество гиперссылок в этом файле (тег ``):

```
>>> links('twolinks.html')
2
```

Изменим функцию `printAddress` с учетом новых сведений:

```
def printAddress(name, line1, line2, city, state, zip, zip2):
    # добавили параметр zip2
    print(name)
    if len(line1) > 0:
        print(line1)
    if len(line2) > 0:
        print(line2)
    # добавили вывод на экран переменной zip2
    print(city + ", " + state + " " + zip + zip2)

# Добавили новый аргумент addr_zip2:
printAddress(addr_name,     addr_line1,     addr_line2,     addr_city,     addr_state,
addr_zip,     addr_zip2)
```

Пришлось несколько раз добавить новый индекс, чтобы функция `printAddress` корректно отработала при новых условиях. Какой недостаток у рассмотренного подхода? Огромное количество переменных! Чем больше сведений о человеке хотим обработать, тем больше переменных мы должны создать. Конечно, можно поместить всё в список (элементами списка тогда будут строки), но в Python есть более универсальный подход для работы с наборами разнородных данных, ориентированный на объекты.

Создадим структуру данных (*класс*) с именем `Address`, которая будет содержать все сведения об адресе человека:

```
class Address: # имя класса выбирает программист
    name = ""      # поля класса
    line1 = ""
    line2 = ""
    city = ""
    state = ""
    zip = ""
```

Класс задает шаблон для хранения адреса. Превратить шаблон в конкретный адрес можно через создание *объекта* (экземпляра)⁴² класса `Address`⁴³:

```
homeAddress = Address()
```

Теперь можем заполнить поля объекта конкретными значениями:

```
# заполняем поле name объекта homeAddress:
homeAddress.name = "Ivan Ivanov"
homeAddress.line1 = "701 N. C Street"
homeAddress.line2 = "Carver Science Building"
homeAddress.city = "Indianola"
homeAddress.state = "IA"
homeAddress.zip = "50125"
```

Создадим еще один объект класса `Address`, который содержит информацию о загородном доме того же человека:

```
# переменная содержит адрес объекта класса Address:
vacationHomeAddress = Address()
```

⁴² В Python классы являются объектами, но для упрощения скажем, что это шаблон

⁴³ Вспомните о создании объекта класса `int`: `a = int()`

```
# self.name - обращение к имени текущего объекта-собаки
print(self.name, " говорит гав")

# Создадим объект myDog класса Dog:
myDog = Dog()

# Присвоим значения полям объекта myDog:
myDog.name = "Spot" # Придумываем имя созданной собаке
myDog.weight = 20    # Указываем вес собаки
myDog.age = 1        # Возраст собаки

# Вызовем метод bark объекта myDog, т.е. попросим собаку подать голос:
myDog.bark()
# Полная форма для вызова метода myDog.bark() будет: Dog.bark(myDog),
# т.е. полная форма требует в качестве первого аргумента сам объект - self
```

Результат работы программы:

```
===== RESTART: C:/Python35-32/ndog.py =====
Spot говорит гав
>>>
```

Данный пример демонстрирует объектно-ориентированный подход в программировании, когда создаются объекты, приближенные к реальной жизни. Между объектами происходит взаимодействие посредством вызова методов. Поля объекта (переменные) фиксируют его состояние, а вызов метода приводит к реакции объекта и/или изменению его состояния (изменению переменных внутри объекта).

Упражнение 15.1

Создайте класс Cat. Определите атрибуты name (имя), color (цвет) и weight (вес). Добавьте метод под названием meow («мяуканье»). Создайте объект класса Cat, установите атрибуты, вызовите метод meow.

В предыдущем примере между созданием объекта myDog класса Dog и присвоению ему имени (myDog.name="Spot") прошло некоторое время. Может случиться так, что программист забудет указать имя и тогда собака будет безымянная – такого допустить мы не можем! Избежать подобной ошибки позволяет специальный метод (конструктор), который вызывается сразу в момент создания объекта заданного класса.

Сначала рассмотрим работу конструктора в общем виде:

```
class Dog:
    name = ""
    # Конструктор вызывается в момент создания объекта этого типа;
    # специальный метод Python, поэтому два нижних подчеркивания
    def __init__(self):
        print("Родилась новая собака!")

# Создаем собаку (объект myDog класса Dog)
myDog = Dog()
```

Запустим программу:

```
>>>
===== RESTART: C:/Python35-32/dog1.py =====
Родилась новая собака!
>>>
```

```
>>>
=====
RESTART: C:/Python35-32/dog3.py =====
Spot
Sharik
>>>
```

Упражнение 15.2

1. Напишите код, описывающий класс Animal:

- добавьте атрибут имени животного
- добавьте метод eat, выводящий «Ням-ням»
- добавьте методы getName и setName
- добавьте метод makeNoise, выводящий «Имя животного говорит Гррр»
- добавьте конструктор классу Animal, выводящий «Родилось животное имя животного»

2. Основная программа:

- создайте животное, в момент создания определите его имя
- узнайте имя животного через вызов метода getName
- измените имя животного через вызов метода setName
- вызовите eat и makeNoise для животного

Упражнение 15.3

Создайте класс StringVar для работы со строковым типом данных, содержащий методы set и get. Метод set служит для изменения содержимого строки, get – для получения содержимого строки. Создайте объект типа StringVar и протестируйте его методы.

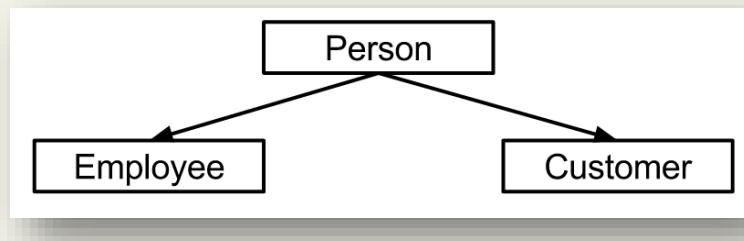
Упражнение 15.4

Создайте класс точки Point, позволяющий работать с координатами (x, y). Добавьте необходимые методы класса.

требует меньшего времени для ответа, чем вопросы «Имеет ли страус крылья?» или «Может ли страус дышать?».

Упомянутое выше свойство наследования нашло свое отражение в объектно-ориентированном программировании.

К примеру, необходимо создать программу, содержащую описание классов Работника (Employee) и Клиента (Customer). Эти классы имеют общие свойства, присущие всем людям, поэтому создадим *базовый* класс Человек (Person) и наследуем от него *дочерние* классы Employee и Customer:



Код, описывающий иерархию классов, представлен ниже:

```
class Person:  
    name = "" # имя у любого человека  
  
class Employee(Person):  
    job_title = "" # наименование должности работника  
  
class Customer(Person):  
    email = "" # почта клиента
```

Создадим объекты на основе классов и заполним их поля:

```
johnSmith = Person()  
johnSmith.name = "John Smith"  
  
janeEmployee = Employee()  
janeEmployee.name = "Jane Employee" # поле наследуется от класса Person  
janeEmployee.job_title = "Web Developer"  
  
bobCustomer = Customer()  
bobCustomer.name = "Bob Customer" # поле наследуется от класса Person  
bobCustomer.email = "send_me@spam.com"
```

В объектах классов Employee и Customer появилось поле name, унаследованное от класса Person.

```
class Person:  
    name = ""  
    def __init__(self):  
        print("Создан человек")  
  
class Employee(Person):  
    job_title = ""  
    def __init__(self):  
        Person.__init__(self) # вызываем конструктор базового класса  
        print("Создан работник")  
  
class Customer(Person):  
    email = ""  
    def __init__(self):  
        Person.__init__(self) # вызываем конструктор базового класса  
        print("Создан покупатель")  
  
johnSmith = Person()  
janeEmployee = Employee()  
bobCustomer = Customer()
```

Результат работы программы:

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

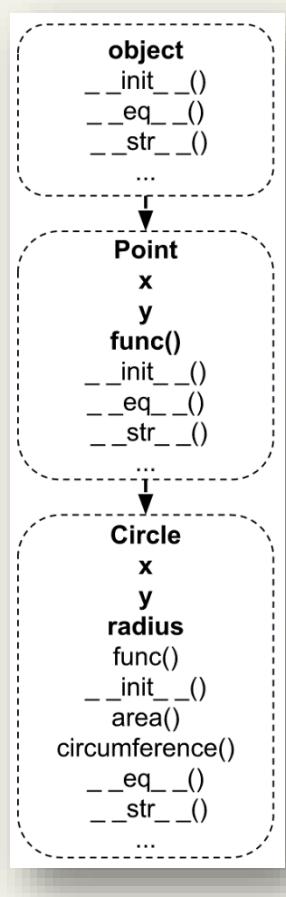
Упражнение 15.5

1. Напишите код, описывающий класс Animal:
 - a) Добавьте атрибут имени животного.
 - b) Добавьте метод eat, выводящий «Ням-ням».
 - c) Добавьте методы getName и setName.
 - d) Добавьте метод makeNoise, выводящий «Имя животного говорит Гррр».
 - e) Добавьте конструктор класса Animal, выводящий «Родилось животное».
2. Пусть Animal будет родительским для класса Cat. Метод makeNoise класса Cat выводит «Имя животного говорит Мяу». Конструктор класса Cat выводит «Родился кот», а также вызывает родительский конструктор.
3. Пусть Animal будет родительским для класса Dog. Метод makeNoise для Dog выводит «Имя животного говорит Гав». Конструктор Dog выводит «Родилась собака», а также вызывает родительский конструктор.
4. *Основная программа.* Код, создающий кота, двух собак и одно простое животное. Дайте имя каждому животному (через вызов методов). Код, вызывающий eat и makeNoise для каждого животного.

Получается, что за всеми операциями над объектами стоят вызовы соответствующих методов. За каждой стандартной операцией над объектами закреплен собственный специальный метод (при сложение вызывается метод `__add__` и т.д.).

Заметим, что мы не переопределяли специальный метод (`__ne__`) для неравенства `a != b`, но Python смог выполнить сравнение, т.к. принял его результат за обратный к равенству (вызов метода `__eq__`).

Наследуем от класса `Point` класс `Circle`:



Исходный код класса `Circle`:

```

import math

class Circle(Point):
    def __init__(self, radius, x=0, y=0):
        super().__init__(x, y) # вызов конструктора базового класса
        self.radius = radius
    def area(self): # площадь окружности
        return math.pi * (self.radius ** 2)
    def circumference(self): # длина окружности
        return 2 * math.pi * self.radius
    def __eq__(self, other): # сравнение двух окружностей
        return self.radius == other.radius and super().__eq__(other)
    def __str__(self): # вывод информации в виде строки
        return "({0.radius}, {0.x}, {0.y})".format(self)

circle = Circle(2) # создаем объект, radius=2, x=0, y=0
circle.radius = 3
circle.x = 12
a = Circle(4, 5, 6)
  
```

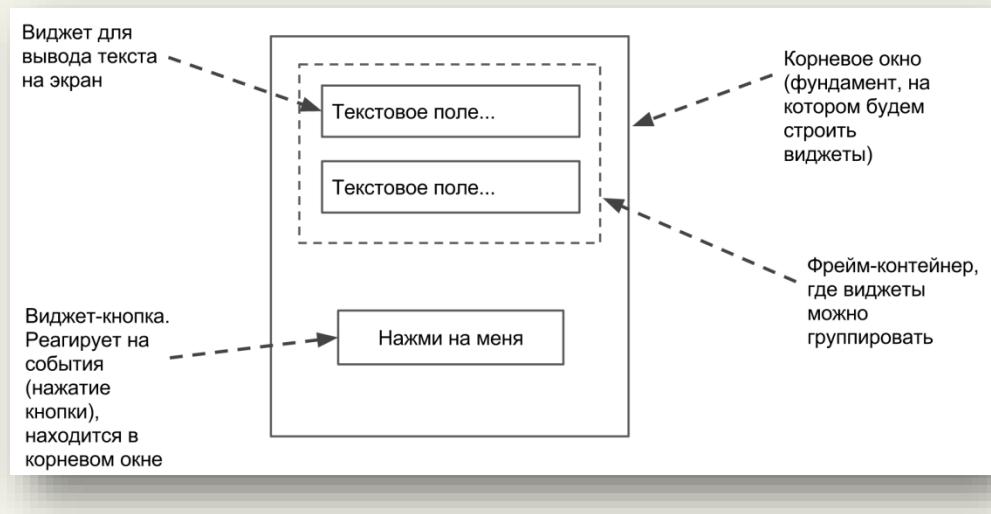
ГЛАВА 16. РАЗРАБОТКА ПРИЛОЖЕНИЙ С ГРАФИЧЕСКИМ ИНТЕРФЕЙСОМ

16.1. Основы работы с модулем *tkinter*

Язык Python позволяет создавать приложения с графическим интерфейсом, для этого используются различные графические библиотеки. Остановимся на рассмотрении стандартной (входит в стандартный комплект Python) графической библиотеки *tkinter*.

Первым делом при работе с *tkinter* необходимо создать главное (корневое) окно, в котором размещаются остальные графические элементы – виджеты. Существуют различные виджеты⁴⁶ на все случаи жизни: для ввода текста, вывода текста, выпадающее меню и пр. Некоторые виджеты (фреймы) используются для группировки других виджетов внутри себя. Есть специальный виджет кнопка, при нажатии на который происходят некоторые события (события можно обрабатывать).

Схематично главное окно с набором виджетов изображено на следующей схеме:



В отдельном файле (*mytk1.py*, но не с именем *tkinter.py!*) выполним следующую простейшую программу для отображения главного окна:

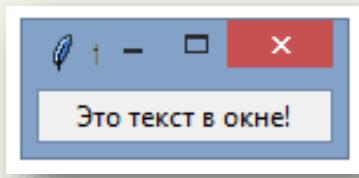
```
# Подключаем модуль, содержащий методы для работы с графикой
import tkinter
# Создаем главное (корневое) окно,
# в переменную window записываем ссылку на объект класса Tk
window = tkinter.Tk()
# Задаем обработчик событий для корневого окна
window.mainloop()
```

⁴⁶ [Перечень виджетов](#)

Следующий пример демонстрирует создание виджета Label:

```
import tkinter
window = tkinter.Tk()
# Создаем объект-виджет класса Label в корневом окне window
# text - параметр для задания отображаемого текста
label = tkinter.Label(window, text="Это текст в окне!")
# Отображаем виджет с помощью менеджера pack
label.pack()
window.mainloop()
```

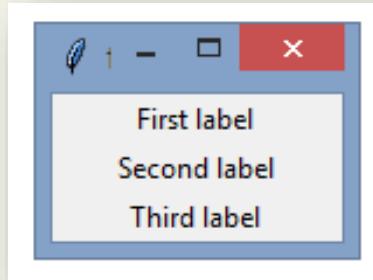
Результат работы программы:



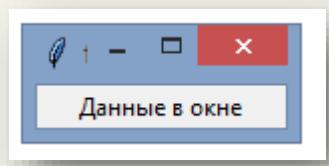
Следующий пример демонстрирует размещение виджетов во фрейме:

```
import tkinter
window = tkinter.Tk()
# Создаем фрейм в главном окне
frame = tkinter.Frame(window)
frame.pack()
# Создаем виджеты и помещаем их во фрейме frame
first = tkinter.Label(frame, text='First label')
# Отображаем виджет с помощью менеджера pack
first.pack()
second = tkinter.Label(frame, text='Second label')
second.pack()
third = tkinter.Label(frame, text='Third label')
third.pack()
window.mainloop()
```

Пример выполнения программы:



Результат выполнения программы:

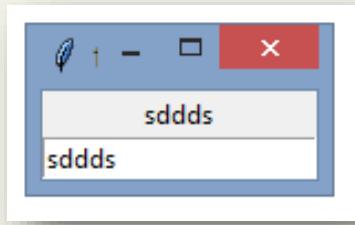


16.2. Шаблон «Модель-вид-контроллер» на примере модуля *tkinter*

Следующий пример показывает, каким образом использовать виджет (`Entry`) для ввода данных:

```
import tkinter
window = tkinter.Tk()
frame = tkinter.Frame(window)
frame.pack()
var = tkinter.StringVar()
# Обновление содержимого переменной происходит в режиме реального времени
label = tkinter.Label(frame, textvariable=var)
label.pack()
# Пробуем набрать текст в появившемся поле для ввода
entry = tkinter.Entry(frame, textvariable=var)
entry.pack()
window.mainloop()
```

Запустим программу и попробуем набрать произвольный текст:

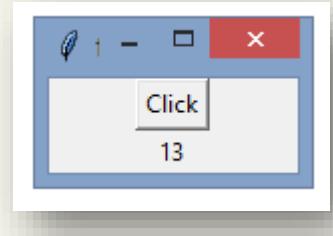


Видим, что текст, который мы набираем, мгновенно отображается в окне. Дело в том, что виджеты `Label` и `Entry` используют для вывода и ввода текста соответственно одну и ту же переменную `var` класса `StringVar`. Подобная схема работы оконного приложения укладывается в универсальный шаблон (паттерн), который называется «Модель-вид-контроллер» (Model-View-Controller или MVC)⁴⁹.

В общем виде под *моделью* (Model) понимают способ хранения данных, т.е. как данные хранятся (например, в переменной какого класса). *Вид* (View) служит для отображения данных. *Контроллер* (Controller) отвечает за обработку данных.

⁴⁹ Паттерн MVC получил широкое распространение при разработке веб-приложений

Результат выполнения программы:



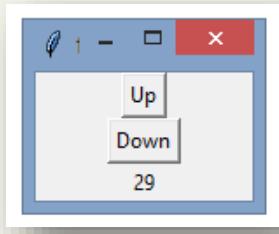
Более сложный пример с двумя кнопками и двумя обработчиками событий (`click_up`, `click_down`):

```
import tkinter
window = tkinter.Tk()
# Модель:
counter = tkinter.IntVar()
counter.set(0)

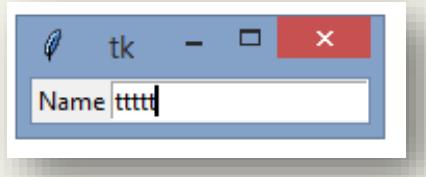
# Два контроллера:
def click_up():
    counter.set(counter.get() + 1)
def click_down():
    counter.set(counter.get() - 1)

# Вид:
frame = tkinter.Frame(window)
frame.pack()
button = tkinter.Button(frame, text='Up', command=click_up)
button.pack()
button = tkinter.Button(frame, text='Down', command=click_down)
button.pack()
label = tkinter.Label(frame, textvariable=counter)
label.pack()
window.mainloop()
```

Результат работы программы:

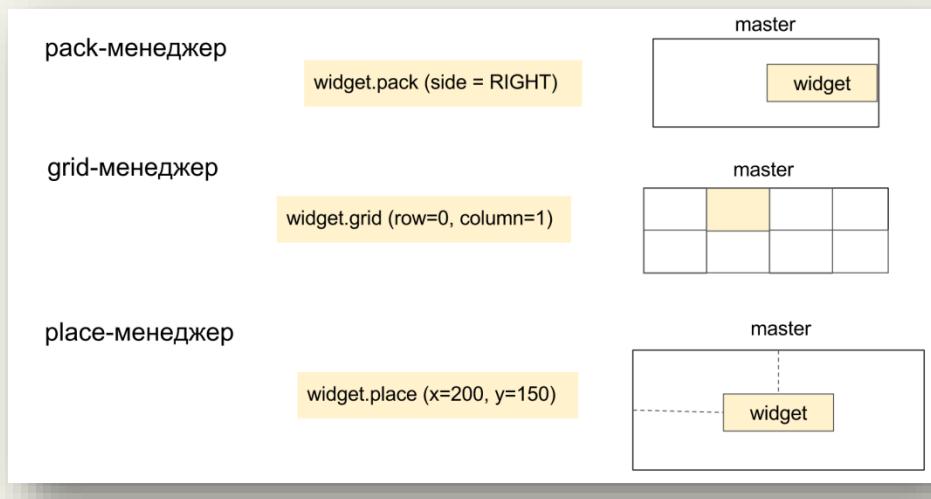


Результат выполнения программы:



Для справки. Менеджеры расположения (геометрии)

Tkinter имеет несколько способов для размещения виджетов. Среди них: *pack*-менеджер, который мы использовали ранее, *grid*-менеджер для задания строки и столбца для размещения виджета и *place*-менеджер для задания координат расположения виджета:



Особенность следующего примера в том, что введенный текст (через виджет `Entry`) отображается на экране (через виджет `Label`) только в момент нажатия кнопки (виджет `Button`), а не в реальном времени, как это было раньше:

```
import tkinter
# Вызывается в момент нажатия на кнопку:
def click():
    # Получаем строковое содержимое поля ввода с помощью метода get
    # С помощью config можем изменить отображаемый текст
    label.config(text=entry.get())

window = tkinter.Tk()
frame = tkinter.Frame(window)
frame.pack()
entry = tkinter.Entry(frame)
entry.pack()
label = tkinter.Label(frame)
label.pack()
# Привязываем обработчик нажатия на кнопку к функции click
button = tkinter.Button(frame, text='Печать!', command=click)
button.pack()
window.mainloop()
```

ГЛАВА 17. КЛИЕНТ-СЕРВЕРНОЕ ПРОГРАММИРОВАНИЕ В PYTHON

Python содержит стандартные модули⁵⁰, позволяющие обратиться к удаленному веб-серверу по протоколу прикладного уровня HTTP, например:

```
>>> import urllib.request as ur
>>> url = "http://www.ya.ru:80"
>>> conn = ur.urlopen(url)
>>> print(conn)
<http.client.HTTPResponse object at 0x0000003E9CD98630>
>>> data = conn.read()
>>> print(data)
b'<!DOCTYPE html><html class="i-ua_js_no i-ua_css_standart i-ua_browser_i-ua_browser_desktop" lang="ru"><head xmlns:og="http://ogp.me/ns#><meta http-equiv="X-UA-Compatible" content="IE=edge"><title>\xd0\xaf\xd0\xbd\xd0\xb4\xd0\xb5\xd0\xba\xd1\x81</title><meta http-equiv=Content-Type content="text/html; charset=UTF-8"><meta name="description" content="\xd0\xaf\xd0\xbd\xd0\xb4\xd0\xb5\xd0\xba\xd1\x81 \xe2\x80\x94 \xd0\xbe\xd0\xb1\xd0\xbb\xd0\xb5\xd0\xb3\xd1\x87\xd1\x91\xd0\xbd\xd0\xbd\x0\xb0\x...
>>> print(conn.status)
200
>>>
```

Переменная `conn` является объектом класса `HTTPResponse`, метод `read` предоставляет информацию о веб-странице, `status` содержит код статуса HTTP-ответа.

Python заранее не знает, что будет передаваться по сети, поэтому используется специальный тип данных `bytes` (байтовые строки):

```
>>> type(b'foo')
<class 'bytes' at 0x65CFC1D8>
>>> b'foo'.decode('utf-8')
'foo'
>>>
```

Следующая команда, выполненная в командной строке, запускает локальный веб-сервер на порту 8000:

```
$ python -m http.server
```

⁵⁰ Подробнее: <https://docs.python.org/3.6/library/http.client.html>

Сервер (обрабатывает поступающие запросы от клиента):

```
import socket # подключаем модуль для взаимодействия по сети

HOST = '127.0.0.1' # IP-адрес для клиент-серверного обмена на одном ПК
PORT = 50007 # порт идентифицирует программу-сервер на данном ПК
# создается программный сокет с гарантированной (SOCK_STREAM)
# доставкой данных (протокол TCP):
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# сокет привязывается (bind) к IP-адресу и сетевому порту для того,
# чтобы обрабатывать поступающие запросы:
s.bind((HOST, PORT))
# сервер слушает (listen), ожидает входные соединения от клиента:
s.listen(1)
# в момент, когда от клиента поступил запрос на соединение, вызывается
# метод accept, который приводит к созданию нового сокета
# (записывается в переменную conn). Данную операцию можно сравнить с
# поступлением телефонного звонка на коммутатор (listen), который
# перенаправляет звонок к конкретному оператору (accept) и снова
# переходит в режим ожидания:
conn, addr = s.accept() # в переменной addr IP-адрес клиента
print('Connected client')
while 1:
    data = conn.recv(1024) # получение данных от клиента, 1024 байт
    if not data:
        break
    else:
        print('Received[2]: ', data)
        conn.send(data) # отправка данных клиенту
        print('Send[3]: ', data)
conn.close() # закрытие соединения
```

Клиент (устанавливает соединение с сервером):

```
import socket

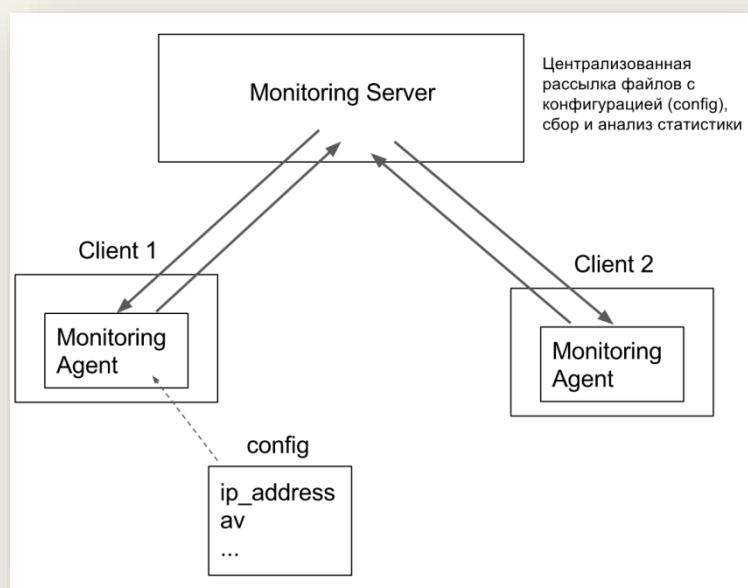
HOST = '127.0.0.1' # IP-адрес сервера
PORT = 50007 # порт сервера
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# клиент устанавливает соединение с сервером:
s.connect((HOST, PORT))
data = 'Hello world'
# обмен по сети происходит в формате bytes, поэтому строку перед
# передачей ее серверу, преобразуем:
s.send(data.encode('utf-8'))
print('Send[1]: ', data)
# получение данных от сервера:
data = s.recv(1024)
s.close()
print('Received[4]: ', data)
```

8 - ответ отправляется клиенту.

Предположим, что реализуется бот-анекдотов. Клиент соединяется с сервером. Отмечу, что клиент предварительно ничего не знает о том, как работать с ботом-анекдотов, т.е. какие существуют команды и пр. В момент соединения с клиентом сервер пересыпает информацию о доступных командах. Например, «/list» - получить список тем анекдотов, «/car» - получить один анекдот на тему автомобилей. Анекдоты хранятся на стороне сервера в текстовом файле (формат файла задается разработчиком). Компоненты шаблона MVC могут быть реализованы в виде отдельных классов либо функций.

Упражнение 17.2

Разработайте распределенную систему мониторинга удаленных хостов:



Каждый хост (операционная система на выбор разработчика) содержит программу-агента, который собирает информацию о текущем состоянии системы, например, контроль запуска определенных служб (контролируемые службы выбираются на усмотрение разработчика, можно реализовать выбор службы для мониторинга через конфигурационный файл). **Необходимо задействовать максимальные возможности Python для работы с операционными системами.**

На хосте производится сбор основных действий агента и результатов мониторинга (время, состояние и пр.). Через определенные интервалы времени агенты отправляют информацию на центральный сервер мониторинга. Сервер мониторинга опрашивает агентов, в ответ получает информацию о текущем состоянии системы. На сервере мониторинга производится логирование основных действий и результатов сбора информации (IP-адрес хоста, время и пр.). Итоговый результат сбора информации представляется в виде таблицы или (желательно) графика.

При реализации системы необходимо задействовать возможности библиотек языка программирования Python (os, xmlrpclib и пр.). В качестве хранилища данных можно использовать текстовые файлы собственного формата, XML-формат, БД (MySQL, SQLite).

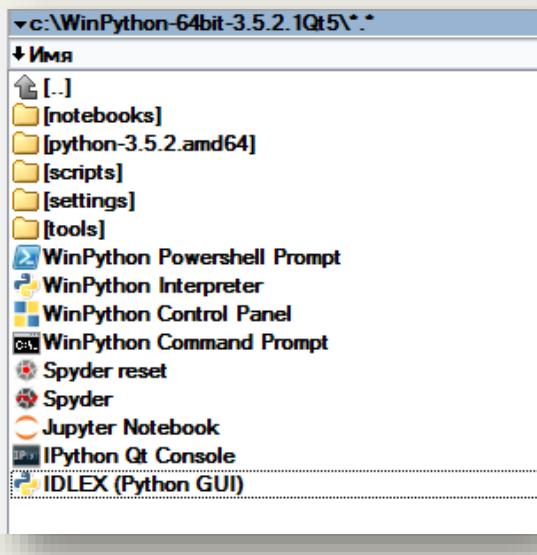
ГЛАВА 18. СРЕДА РАЗРАБОТКИ JUPYTER

Jupyter является развитием проекта IPython, который в интерактивном режиме посредством веб-интерфейса позволяет на языке Python выполнять научные вычисления, строить графики и т.д. Jupyter в отличие от IPython включает в себя не только интерпретатор языка Python, но и поддержку таких языков как Scala, bash, Haskell, Julia, R, Ruby. Выполнить тестовый запуск полноценной версии Jupyter можно на сайте: <https://try.jupyter.org>

18.1. Установка и запуск Jupyter

Для установки Jupyter под ОС Windows понадобится скачать и распаковать дистрибутив WinPython 3.5 (с сайта <https://winpython.github.io>).

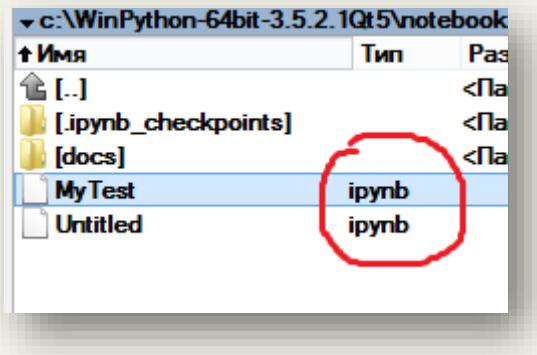
После установки папка с файлами WinPython будет иметь следующий вид:



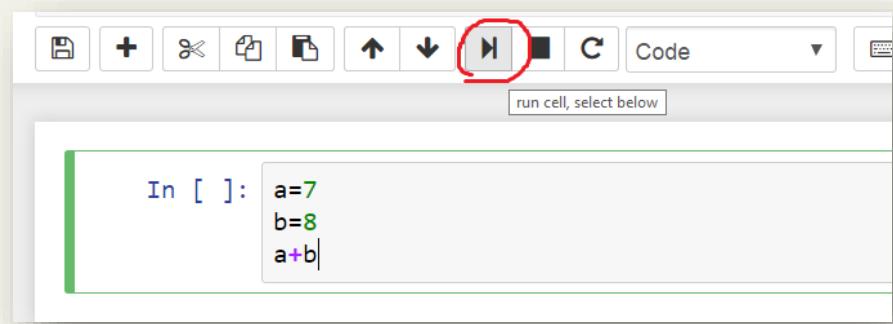
Второй вариант установки Jupyter под ОС Windows (и Linux) – Anaconda (с сайта: <https://www.continuum.io/downloads>).

Вернемся к WinPython. Запустим Jupyter Notebook. В процессе запуска создается локальный веб-сервер, прослушивающий сетевой порт с номером 8888. Автоматически на странице <http://localhost:8888/tree> откроется браузер.

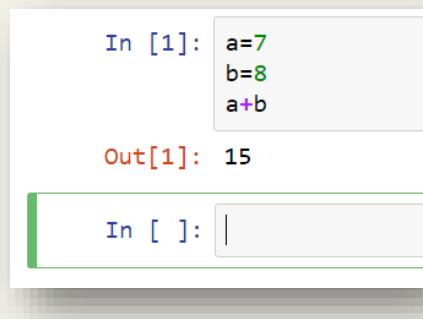
Создадим новый блокнот для запуска программ на языке Python:



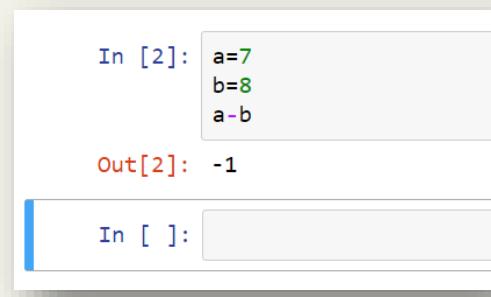
По аналогии с IDLE в ячейке `In []` блокнота Jupyter набираем код на языке Python и запускаем (комбинации $\langle \text{Ctrl} \rangle + \langle \text{Enter} \rangle$, $\langle \text{Alt} \rangle + \langle \text{Enter} \rangle$ – выполнить ячейку и добавить новую ячейку, $\langle \text{Shift} \rangle + \langle \text{Enter} \rangle$ – выполнить ячейку и выделить следующую):



Результат выполнения кода отобразится в ячейке `Out[1]`:



Код можно модифицировать и запустить повторно (изменится индексация ячеек):



```
Type:           list
String form:  [3, 6, 7, 5, 'h', 5]
Length:        6
Docstring:
list() -> new empty list
list(iterable) -> new list initialized from iterable's items
```

Для функций ? показывает строку документации, ?? – по возможности показывает исходный код функции.

3. С помощью «магической» команды %run (получить справочную информацию: %run?) выполнять программы на языке Python.
4. Список «магических» команд %magic .
5. %reset – удаляет все переменные, определенные в интерактивном пространстве имен.
6. Команды для работы с операционной системой:

Таблица 3.3. Команды IPython, относящиеся к операционной системе

Команда	Описание
!cmd	Выполнить команду в оболочке системы
output = !cmd args	Выполнить команду и сохранить в объекте output все выведенное на стандартный вывод
%alias alias_name cmd	Определить псевдоним команды оболочки
%bookmark	Воспользоваться системой закладок IPython
%cd каталог	Сделать указанный каталог рабочим
%pwd	Вернуть текущий рабочий каталог
%pushd каталог	Поместить текущий каталог в стек и перейти в указанный каталог
%popd	Извлечь каталог из стека и перейти в него
%dirs	Вернуть список, содержащий текущее состояние стека каталогов
%dhist	Напечатать историю посещения каталогов
%env	Вернуть переменные среды в виде словаря

18.3. Интерактивные виджеты в Jupyter Notebook

Выполним в Jupyter (IPython) Notebook следующий код:

```
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))
```

ГЛАВА 19. ПРИМЕНЕНИЕ ЯЗЫКА PYTHON

19.1. В области защиты информации и системного администрирования

Упражнение 19.1: разработка генератора стойких паролей (+ пользовательский интерфейс).

В качестве входных параметров генератора можно указать:

1. наличие цифр;
2. наличие прописных букв;
3. наличие строчных букв;
4. наличие спец. символов %, *,),?, @, #, \$, ~
5. длину пароля.

Упражнение 19.2: разработка доброго сетевого шпиона-анализатора.

Система отслеживания, анализа и хранения информации о сетевой активности. Информация о трафике, полученная с помощью снiffeра⁵⁴, анализируется, группируется в события, связывается с различными сущностями (клиенты, сервера) и сохраняется в базе данных.

Далее требования к системе в порядке возрастания сложности.

- Хранение информации⁵⁵ о пакетах, внешних и внутренних IP-адресах. Минимальный веб-интерфейс с отображением статистики.
- Хранение информации о клиентах (браузеры, мессенджеры и т.д.), доменах, IP-адресах и т.д. и их связи между собой. Динамически обновляемая статистика с веб-интерфейсом.
- Продвинутый веб-интерфейс с динамически обновляемой статистикой, возможностями поисковых запросов через веб-интерфейс.

Упражнение 19.3: разработка SIEM

Анализ log-файлов.

19.2. В области искусственного интеллекта



Кадр из фильма «Она» (2013 г.)

⁵⁴ Можно воспользоваться [библиотекой Scapy](#)

⁵⁵ Можно воспользоваться СУБД [MongoDB](#) и модулем [PyMongo](#)

ГЛАВА 20. ПРОГРАММИРОВАНИЕ КОНТРОЛЛЕРА ARDUINO

В следующем примере понадобится контроллер Arduino Mega 2560, модуль датчика освещенности⁵⁷ (LM393) и щилд с ЖК дисплеем (LCD Keypad Shield).

Скетч отображает значение освещенности на ЖК дисплее и передает их через последовательный порт:

```
// include the library code:  
#include <LiquidCrystal.h>  
  
// initialize the library with the numbers of the interface pins  
LiquidCrystal lcd(8, 9, 4, 5, 6, 7);  
  
const int analogSignal = A1; //подключение аналогового сигнального пина  
const int digitalSignal = 7; //подключение цифрового сигнального пина  
boolean noLight; //переменная для хранения значения о присутствии света  
  
int lightness = 0; //переменная для хранения количества света  
  
void setup() {  
    pinMode(digitalSignal, INPUT); //установка режима пина  
    Serial.begin(9600); //инициализация Serial порта  
    // set up the LCD's number of columns and rows:  
    lcd.begin(16, 2);  
}  
  
void loop() {  
    noLight=digitalRead(digitalSignal); //читываем значение о присутствии света  
    lightness=analogRead(analogSignal); // и о его количестве  
    // set the cursor to column 0, line 1  
    lcd.setCursor(0, 0);  
  
    //вывод сообщения  
    Serial.print("There is ");  
    if (noLight) {  
        Serial.println("dark");  
        lcd.print("dark");  
    }  
    else {  
        Serial.println("lightly");  
        lcd.print("lightly");  
    }  
  
    Serial.print("value: ");  
    Serial.println(lightness);  
    lcd.setCursor(0, 1);  
    lcd.print(lightness);  
  
    delay(1000); //задержка 1 сек  
}
```

Результат работы скетча представлен на рисунке:

⁵⁷ [Подробнее](#)

ГЛАВА 21. ИМПОРТИРОВАНИЕ МОДУЛЕЙ, НАПИСАННЫХ НА ЯЗЫКЕ С

Все действия в этой главе производятся в ОС Linux/Debian, поэтому требуется предварительно ее установить.

Для создания модулей на языке С воспользуемся пакетом `distutils`⁵⁸, входящим в состав стандартной библиотеки Python.

Рассмотрим пример⁵⁹ создания собственного Python-модуля на языке С. Для этого нам понадобится создать файл на языке С (`ownmod.c`), представляющий сам модуль⁶⁰:

```
#include <Python.h>

static PyObject* py_echo( PyObject* self, PyObject* args ) {
    printf( "вывод из экспортированного кода!\n" );
    return Py_None;
}

static PyMethodDef ownmod_methods[] = {
    { "echo", py_echo, METH_NOARGS, "echo function" },
    { NULL, NULL }
};

// эта структура добавилась в Python 3:
static struct PyModuleDef ownmodule = {
    PyModuleDef_HEAD_INIT,
    "ownmod", /* name of module */
    NULL,      /* module documentation, may be NULL */
    -1,        /* size of per-interpreter state of the module,
                 or -1 if the module keeps state in global variables. */
    ownmod_methods
};

// Python 2:
//PyMODINIT_FUNC initownmod() {
// Python 3:
PyMODINIT_FUNC PyInit_ownmod() {

    // В Python 2 обходились без создания ownmodule:
    //(void)Py_InitModule( "ownmod", ownmod_methods );

    // Python 3:
    PyObject *m;
    m = PyModule_Create(&ownmodule);
    if (m == NULL)
        return NULL;
}
```

Затем формируем файл `setup.py`:

```
from distutils.core import setup, Extension

module1 = Extension( 'ownmod', sources = [ 'ownmod.c' ] )

setup( name = 'ownmod',
       version = '1.1',
       description = 'This is a first package',
       ext_modules = [module1]
)
```

⁵⁸ Подробнее: <https://docs.python.org/3.6/library/distutils.html>

⁵⁹ Источник примера (для Python 2): https://www.ibm.com/developerworks/ru/library/l-python_details_07/

⁶⁰ Описание структур: <https://docs.python.org/3/c-api/structures.html>

ГЛАВА 22. ДОПОЛНИТЕЛЬНЫЕ МАТЕРИАЛЫ ПО ЯЗЫКУ PYTHON

Online-сервисы и учебники для изучения Python

1. Много интересных и полезных новостей из мира Python: <https://pythondigest.ru/>
2. Игра для обучения программированию: <https://checkio.org>
3. Online IDE: <https://repl.it>
4. Визуализатор online на Python: <http://pythontutor.com/visualize.html#mode=edit>
5. Задания по Python на основе рейтинга: <https://www.hackerrank.com>
6. [Видео лекции «Программирование на языке Python для сбора и анализа данных»](#)
7. [Курс «Программирование на Python \(Институт биоинформатики\)»](#)
8. [Курс «Python: основы и применение» \(Институт биоинформатики\)](#)
9. [Видео лекции «Python 3 Basics Tutorial Series»](#)
10. Курс [Программирование на Python](#) от Mail.Ru Group

Здравствуйте, Дмитрий Юрьевич.

Хочу поблагодарить Вас за учебник по основам программирования. Нахожу Вашу книгу крайне полезной.

С уважением,
Игорь Гелахов, 54 года, инженер (Р. Беларусь, г. Могилев)

Здравствуйте, Дмитрий!

Спасибо вам за учебное пособие, оно подтолкнуло меня на осуществление давно появлявшегося в мыслях намерение познакомиться с языком.

Хотел бы высказать свой взгляд на алгоритм "Поиск, удаление, поиск", код которого приведён на странице 91.

На мой взгляд, более очевидным решением будет не прибавление 1 к индексу второго найденного элемента, а получение этого индекса уже после возвращения первого найденного элемента в список.

Мой вариант такой:

```
def find_two_smallest(lst):
    min1 = min(lst)
    min1_idx = lst.index(min1)
    lst.remove(min1)

    min2 = min(lst)
    lst.insert(min1_idx, min1)
    min2_idx = lst.index(min2)

    print("Min1 idx: {0} val: {1}".format(min1_idx, min1))
    print("Min2 idx: {0} val: {1}".format(min2_idx, min2))

    return (min1, min2)
```

С уважением,
Павел