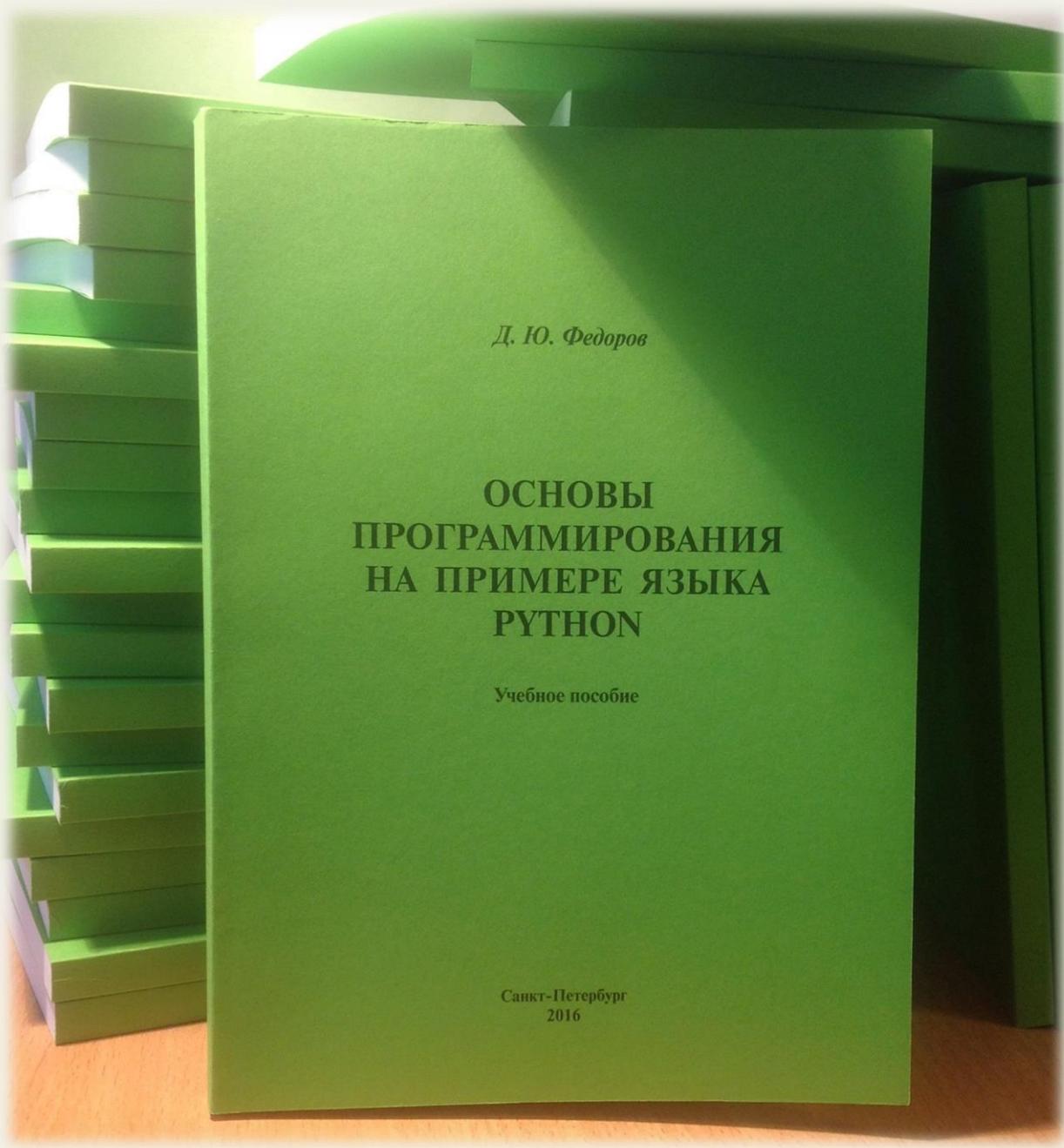


Д. Ю. Федоров

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ПРИМЕРЕ ЯЗЫКА PYTHON

Актуальная версия учебника доступна по ссылке:
<http://dfedorov.spb.ru/python3/book.pdf>

Связаться с автором
(вопросы, предложения, сотрудничество):
dmitriy.fedoroff@gmail.com



Федоров Д. Ю. Основы программирования на примере языка Python :
учеб.пособие / Д. Ю. Федоров. – СПб., 2016. – 176 с.

Оглавление

Предисловие.....	5
1. Основы основ.....	7
2. Знакомство с языком программирования Python.....	11
3. Начинаем программировать в интерактивном режиме	12
3.1. Интеллектуальный калькулятор	12
3.2. Переменные в Python.....	13
3.3. Функции.....	17
4. Программы в отдельном файле.....	21
5. Строки и операции над ними	26
6. Операторы отношений.....	33
7. Условный оператор if	40
8. Модули в Python	44
9. Создание собственных модулей	47
10. Строковые методы в Python	53
11. Списки в Python	58
11.1. Создание списка	58
11.2. Операции над списками	60
11.3. Пседонимы и клонирование списков	65
11.4. Методы списка.....	67
11.5. Преобразование типов	68
11.6. Вложенные списки	69
12. Операторы цикла в Python	70
12.1. Оператор цикла for	70
12.2. Функция range()	73
12.3. Создание списка	75
12.4. Оператор цикла while	78
12.5. Вложенные циклы	81
13. Множества.....	83
14. Кортежи	86
15. Словари.....	87
16. Несколько слов об алгоритмах.....	90
17. Обработка исключений в Python.....	95

18. Работа с файлами в Python.....	98
19. Объектно-ориентированное программирование в Python	108
19.1 Основы объектно-ориентированного подхода	108
19.2. Наследование в Python	115
19.3. Иерархия наследования в Python	119
20. Разработка приложений с графическим интерфейсом	122
20.1. Основы работы с модулем tkinter	122
20.2. Шаблон «Модель-вид-контроллер» на примере модуля tkinter.....	126
20.3. Изменение параметров по умолчанию при работе с tkinter	129
21. Клиент-серверное программирование в Python.....	132
Вопросы к зачету по языку программирования Python (базовый уровень).....	137
22. Продвинутый уровень Python.....	138
22.1. Функциональный подход.....	138
22.2. Импортирование модулей, написанных на языке С (для Python 3).....	138
23. Python и веб-программирование на примере фреймворка Flask.....	141
24. Jupyter (IPython). Расширенные возможности Python.....	142
24.1. Установка и запуск Jupyter (IPython)	142
24.2. Работа в Jupyter (IPython).....	146
24.3. Интерактивные виджеты в Jupyter (IPython) Notebook.....	147
24.4. Установка дополнительных пакетов в WinPython из PyPI	148
25. Применение Jupyter (IPython) в области защиты информации и системного администрирования	149
26. Применение Jupyter (IPython) в области анализа данных (искусственного интеллекта)	
.....	149

Предисловие

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

(1981 год, академик А.П. Ершов)

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

(1980 год, профессор Сеймур Пейперт)

В основу предлагаемого учебного пособия положен цикл видео-уроков¹ и занятий, проведенных автором для студентов СПбГЭУ, учеников лицея № 95 и слушателей курсов Epic Skills.

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

Язык программирования Python входит в пятерку² по популярности в мире, поэтому найти по нему литературу не составит труда. На желающих стать программистами обрушится гора справочников и «лучших рекомендаций» по разработке приложений любого уровня сложности, но среди всех этих книг новичку бывает сложно разобраться, а первое знакомство с толстыми справочниками по внутреннему устройству Python может навсегда отпугнуть от занятия программированием.

На взгляд автора, не следует сваливать на головы учащихся сразу всю справочную информацию и множество правил, существующих в языках программирования. «Не следует множить сущее без необходимости»³. Некоторые темы в пособии специально пришлось упростить, чтобы в вводном курсе не вдаваться в излишние детали, но в век Интернета поиск справочной информации не должен составить труда.

Автор благодарит всех, кто принял участие в разработке данного курса.

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

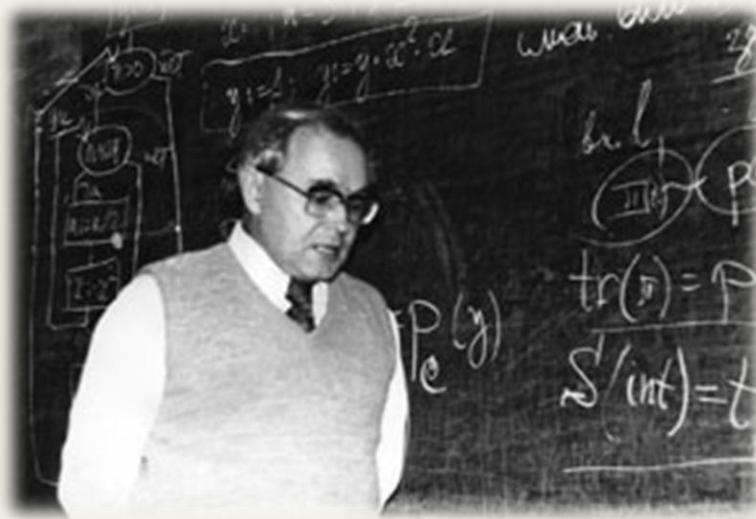


¹ [Python: быстрый старт](#)

² [TIOBE Index for March 2016](#)

³ [Бритва Оккама](#)

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 Пейпера

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

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

1. Основы основ

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

(Эдсгер Дейкстра)

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

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

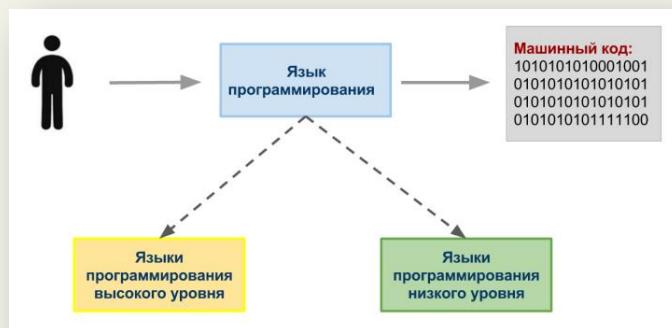
Для того чтобы научиться плавать необходимо войти в воду и начать пробовать грести руками, помогая себе ногами, затем поборов страх, оторваться от дна и поплыть. Есть в этом сходство с написанием программ. Можно прочесть толстый учебник, сдать зачет/экзамен в вузе, но при этом не научиться написанию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

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

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

Компьютер способен работать только с двумя видами сигналов: 1 или 0 (машинным кодом). Писать программы вида 10101010100101010 для человека сложно, мышление его устроено иначе, поэтому появились программы-трансляторы с языка программирования, понятного человеку, на машинный язык, понятный компьютеру.

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



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 С, 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. В последствии операционная система UNIX получила широкое распространение (в наши дни больше известны ее клоны GNU/Linux), а вместе с ней – появилось множество программистов, для которых язык С стал родным.

Написание программ на этом языке требует хорошей квалификации от программиста, т.к. незамеченная ошибка способна привести к серьезным последствиям в работе программы. До сих пор язык С лидирует в качестве языка для системного программирования.

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

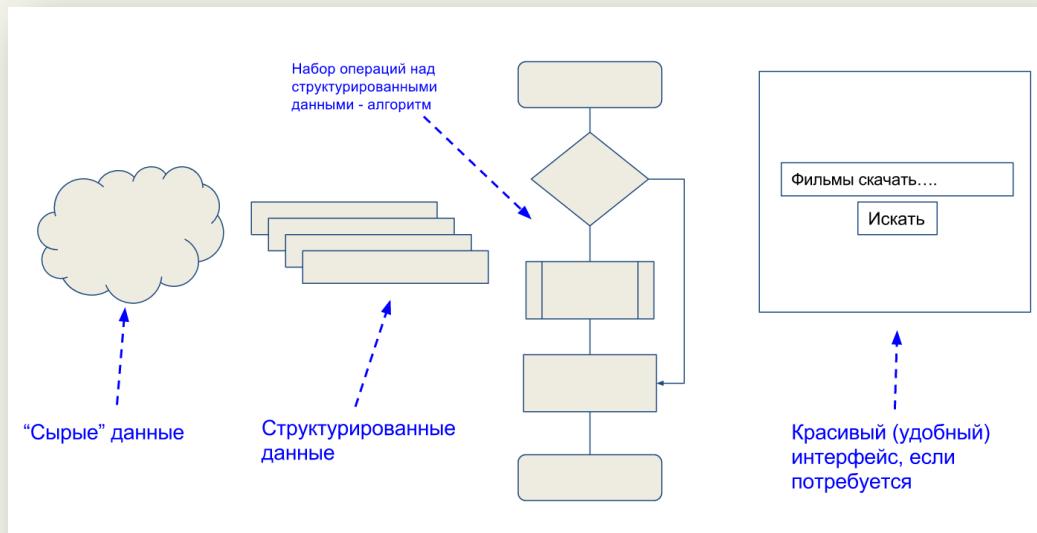
В 90-ые годы появляются персональные компьютеры и сеть Интернет, потому требуются новые технологии и языки программирования. В этот момент набирает популярность язык Java, который позволяет в кратчайшие сроки начать писать крупные приложения без опасений что-либо серьезно испортить в системе. Язык Java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

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

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

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

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. 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.

2. Знакомство с языком программирования Python

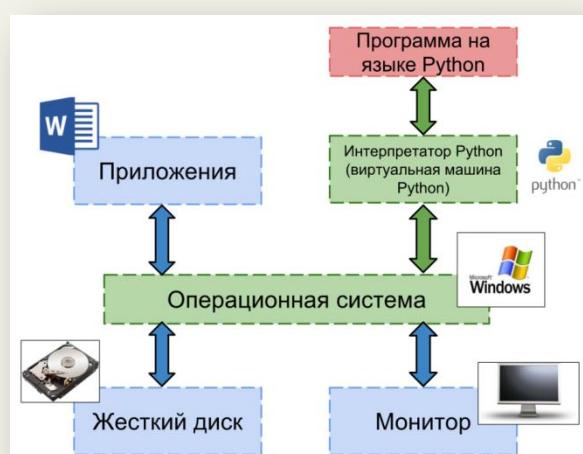
Чтобы читатель не подумал, что Python – игрушечный язык программирования, на котором можно только обучать основам программирования, кратко перечислю области, где он активно используется:



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

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

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



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

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

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

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

3. Начинаем программировать в интерактивном режиме

3.1. Интеллектуальный калькулятор

В самом начале обучения Python можно рассматривать как обычный интерактивный калькулятор. В интерактивном режиме IDLE найдем значения следующих математических выражений. 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.

В математических выражениях в качестве операндов можно использовать целые числа⁴ (1, 4, -5) или вещественные⁵ (в программировании их еще называют числами с

⁴ А также комплексные числа, логические значения (True, False)

⁵ Подробное объяснение правил хранения вещественных чисел в компьютере выходит за рамки нашего курса, сравните в следующем примере результаты вычислений:

```
>>> 2/3+1  
1.6666666666666665  
>>> 5/3
```

плавающей точкой): 4.111, -9.3. Математические операции, доступные над числами в Python⁶:

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

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

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

В качестве упражнения найдите значение выражения $2+56*5.0-45.5+5^5$. При вычислении математических выражений Python придерживается приоритета операций:

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

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

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

3.2.Переменные в Python

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:

```
1.6666666666666667
>>>
```

⁶ Выражение $(b * (a // b)) + a \% b$ эквивалентно a .

```
>>> 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	exec
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:

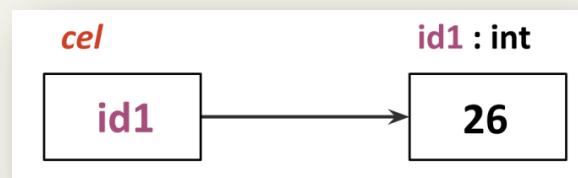
$$T_F = 9/5 * T_C + 32$$

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 – полностью объектно-ориентированный язык программирования. В чем это выражается?



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

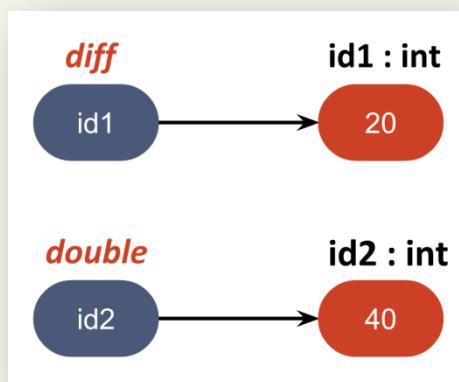
Вычисление следующего выражения в итоге приведет к присваиванию переменной `cel` значения 72, т.е. сначала вычисляется правая часть, затем результат присваивается левой части.

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

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

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

По окончании вычислений память для Python будет иметь следующий вид:



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

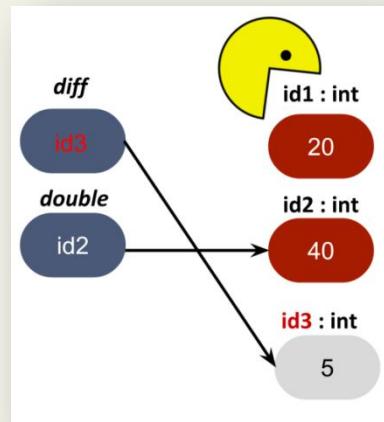
```
>>> diff = 5
>>> double
40
>>> diff
5
>>>
```

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

Возможно, читатель заметил, что Python не изменяет существующие числовые объекты, а создает новые. 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
>>>
```



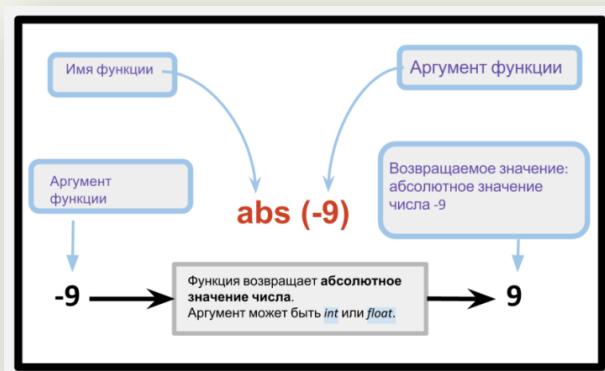
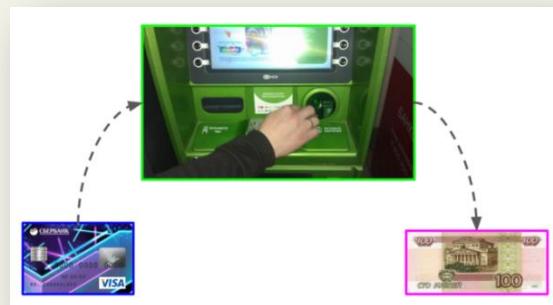
Если вспомнить, что сначала вычисляется правая часть, то все легко объясняется.

3.3. Функции

Функция в Python является фундаментом 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
>>> abs (-9) + abs (5.6)
14.6
>>>
  
```

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

Рассмотрим несколько популярных математических функций языка Python. `pow(x, y)` возвращает значение `x` в степени `y`. Эквивалентно записи `x**y`.

```

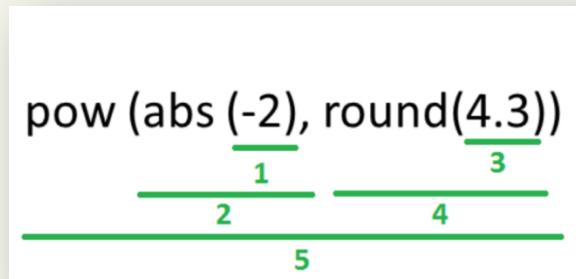
>>> pow(4, 5)
1024
>>>
  
```

`round(number)` возвращает число с плавающей точкой, округленное до 0 цифр после запятой (по умолчанию).

Может быть вызвана с двумя аргументами: `round(number[, ndigits])`, где `ndigits` – число знаков после запятой.

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

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



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

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

`int()` возвращает целочисленный объект, построенный из числа или строки⁷, или 0, если аргументы не переданы.

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

Рассмотрим примеры:

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

⁷ Об этом типе данных в следующем разделе

В качестве упражнения найдите следующие значения:

```
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.
```

>>>

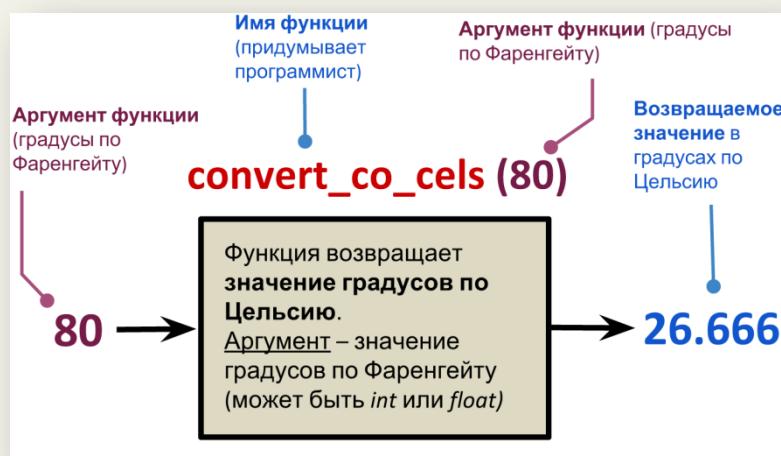
Вернемся к формуле перевода градусов по шкале Фаренгейта (T_F) в градусы по шкале Цельсия (T_C):

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

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)
```

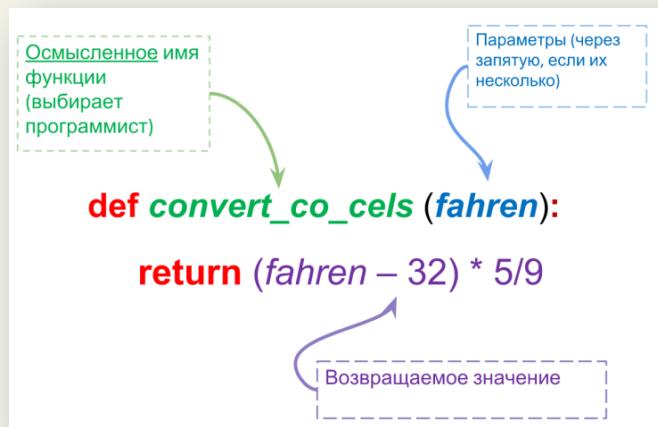
Заметим, что каждый раз нам приходится набирать одну и ту же формулу для перевода. Упростим наши вычисления, создав собственную функцию, переводящую градусы по Фаренгейту в градусы по Цельсию.



В первую очередь, необходимо придумать имя функции, к примеру, назовем функцию `convert_to_cels`. Постарайтесь, чтобы имя было осмысленным (`lena123` – плохой пример для имени функции) и отражало смысл функции, вспомните о правилах

наименования переменных. Помимо этого, не желательно, чтобы имя вашей функции совпадало с именами встроенных функций Python (встроенные функции в IDLE подсвечиваются фиолетовым цветом).

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



Перейдем непосредственно к созданию функции. Ключевое слово `def` для Python означает, что дальше идет имя создаваемой функции. После `def` указывается имя функции `convert_co_cels`, затем в скобках указывается параметр, которому будет присваиваться значение при вызове функции. Параметры функции – обычные переменные, которыми функция пользуется для внутренних вычислений. Переменные, объявленные внутри функции, называются *локальными* и не видны вне функции. После символа «`:`» начинается тело функции. В интерактивном режиме Python самостоятельно поставит отступ⁸ от края экрана, тем самым обозначив, где начинается тело функции. Выражение, стоящее после ключевого слова `return` будет возвращаться в качестве результата вызова функции.

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

```
>>> def convert_co_cels(fahren):  
    return (fahren-32)*5/9  
  
>>> convert_co_cels(451)  
232.77777777777777  
>>> 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:

$$\begin{matrix} x^4 & + & 4^x \\ y^4 & + & 4^x \end{matrix}$$

⁸ Отступы играют важную роль в Python, отделяя тело функции, цикла и пр.

4. Программы в отдельном файле

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.

Создадим файл myprog.py, содержащий следующий код (тело функции должно отделяться либо четырьмя пробелами, либо одной табуляцией – придерживайтесь одного из этих правил):

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

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

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

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

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

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

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

print(f(4))      # комментарии игнорируются Python
print(f(56))
```

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

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

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

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 строка⁹.

После выполнения программы получим следующий результат:

⁹ О строках подробно поговорим в следующей главе

```
>>>
=====
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 объявления `global`:

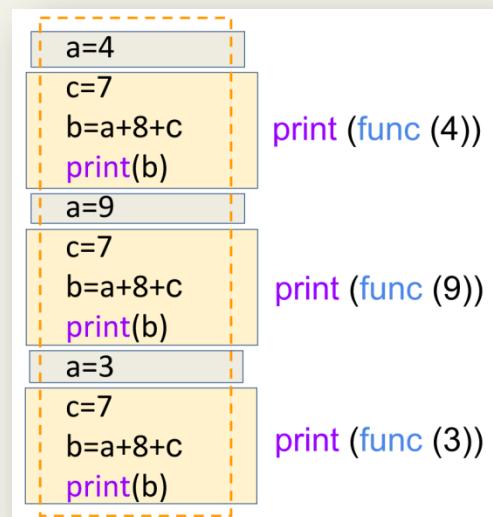
```
x = 50 # глобальная переменная
def func():
    global x # указываем, что x-глобальная переменная
    print('x равно', x)
    x = 2      # изменяем глобальную переменную
    print('Заменяем глобальное значение x на', x)

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

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

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

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

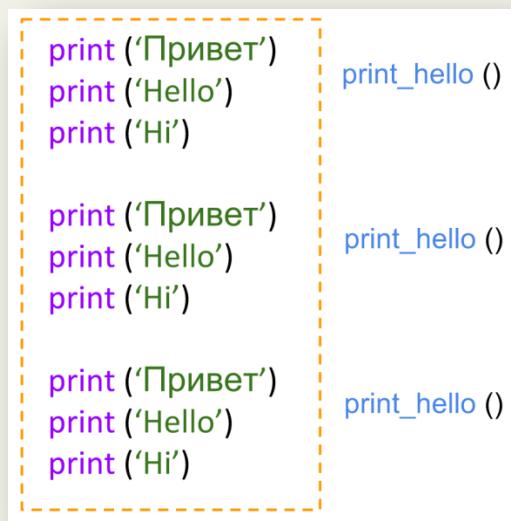


В файле не забываем вызывать функцию `print()`.

Бывают случаи, когда наша функция ничего не принимает на вход и ничего не возвращает¹⁰ (не используется ключевое слово `return`). Пример подобной функции:

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

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



Упражнение 4.1

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 4.2

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

Упражнение 4.3

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

¹⁰ На самом деле, если не указать `return`, то Python вернет объект `None`.

Для справки

Рассмотрим несколько полезных особенностей при работе с функциями в Python. Имена функций в Python являются переменными, содержащими адрес объекта¹¹ типа функция¹², 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:

```
def summa (x, y):  
    return x + y  
  
def func (f, a, b):  
    return f (a, b)  
  
v = func (summa, 10, 3) # передаем summa в качестве аргумента
```

Этот пример демонстрирует, как из функции `func ()` можно вызывать функцию `summa ()`.

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

```
def func(a, b=5, c=10):  
    print('a равно', a, ', b равно', b, ', a с равно', c)  
  
func(3, 7) # a=3, b=7, c=10  
func(25, c=24) # a=25, b=5, c=24  
func(c=50, a=100) # a=100, b=5, c=50
```

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

Упражнение 4.4

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

¹¹ В Python все является объектами.

¹² Да, да, это еще один тип данных.

5. 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!'  
>>>
```

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

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

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

```
>>> 'Марс' + str(5)
'Mарс5'
>>>
```

Можно ли выполнить обратное преобразование типов? Можно!

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

Попросим Python повторить нашу строку заданное число раз:

```
>>> "СПАМ" * 10
'СПАМСПАМСПАМСПАМСПАМСПАМСПАМСПАМ'
>>>
```

Операция умножения для строк приобрела другой смысл.

Строки можно присваивать переменным и дальше работать с переменными:

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

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

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

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

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	- переход 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\\пстрока'  
>>>
```

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\\пстрока')  
Это длинная  
строка  
>>>
```

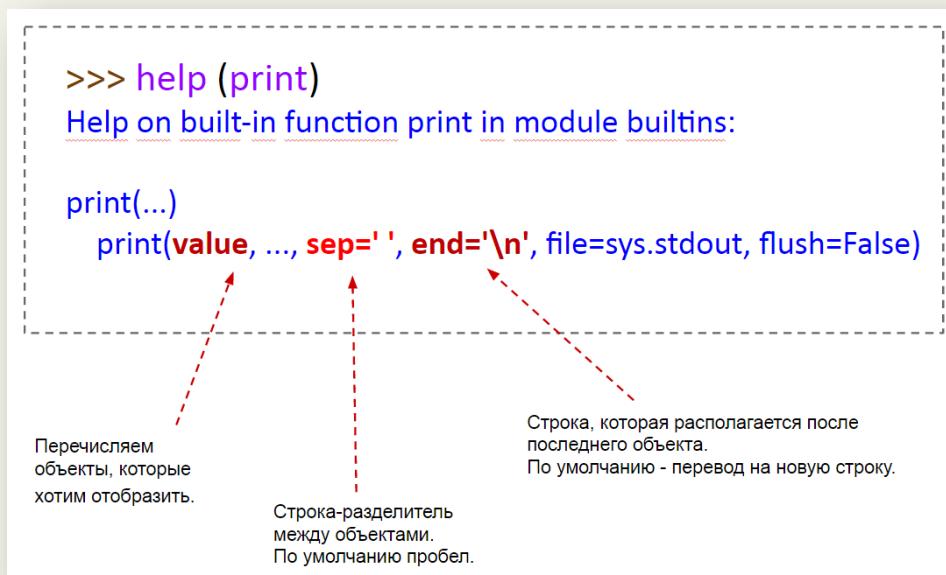
Функция `print()` специальный символ смогла распознать и сделать перевод строки.

Рассмотрим еще несколько примеров:

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

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

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



И снова несколько примеров:

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

По умолчанию `print()` использует в качестве разделителя пробел, мы можем изменить разделитель, изменив для этого принудительно значение параметра `sep=':'`. Отсюда вывод: полезно читать документацию.

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

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

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 5.1

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.

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

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

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

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

Какая ситуация с изменением строк в Python?

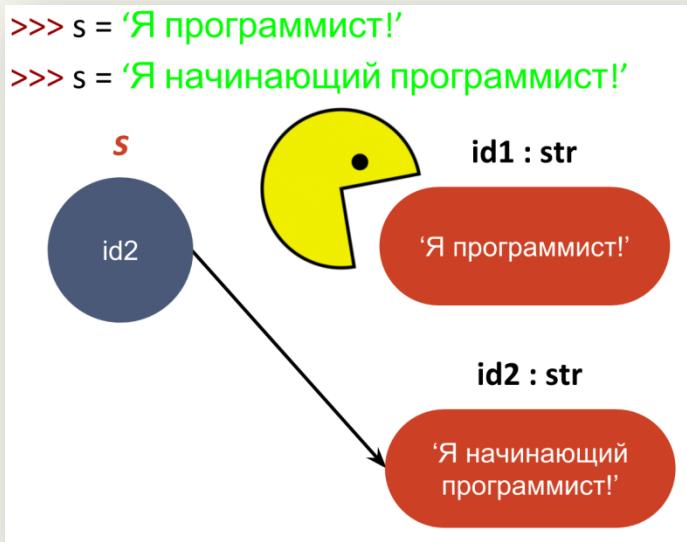
```
>>> s = 'Я люблю писать программы!'
>>> s[0]='Ж'
Traceback (most recent call last):
  File "<pyshell#41>", line 1, in <module>
    s[0]='Ж'
TypeError: 'str' object does not support item assignment
>>>
```

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

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



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'
```

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

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

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

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

Упражнение 5.2

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

Пример работы программы:

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

6. 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`¹³(Истина == 1¹⁴), когда сравнение верное и `False` (Ложь == 0) – в ином случае. `True` и `False` относятся к логическому (булевому) типу данных `bool`.

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

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

Логическим высказыванием (предикатом)¹⁵ будем называть любое повествовательное предложение, в отношении которого можно однозначно сказать, истинно оно или ложно.

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

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

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

«Петров – врач И шахматист». Это высказывание истинно, если оба высказывания «Петров – врач» и «Петров – шахматист» являются истинными.

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

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

```
>>> 2>4
False
>>> 45>3
True
>>> 2>4 and 45>3    # комбинация False and (И) True вернет False
False
>>> 2>4 or 45>3     # комбинация False or (ИЛИ) True вернет True
True
>>>
```

Все, что мы сказали про комбинацию логических высказываний, можно объединить и представить в виде таблицы¹⁶, где 0 – `False`, а 1 – `True`.

¹³ Важно писать в большой буквы

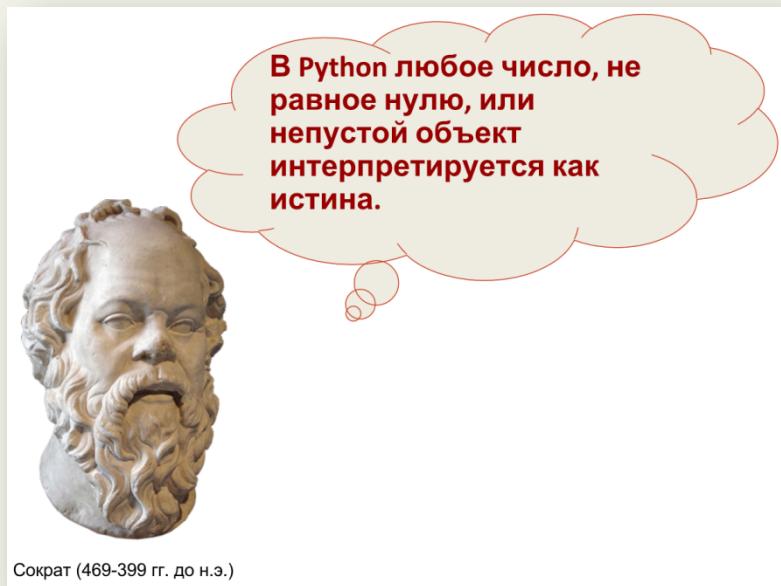
¹⁴ `True` интерпретируется Python как число 1, а `False` как число 0

¹⁵ Подробнее здесь: http://book.kbsu.ru/theory/chapter5/1_5_1.html

¹⁶ Ее называют таблицей истинности

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

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



В Python любое число, не равное нулю, или непустой объект интерпретируется как истина.

Числа, равные нулю, пустые объекты и специальный объект `None`¹⁷ интерпретируются как ложь.

Рассмотрим пример:

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

¹⁷ Он имеет тип `NoneType`

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
>>>
```

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

Логический оператор `and` (И) вернет `True` (истину) или `False` (ложь)¹⁸, если его operandами являются логические высказывания.

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

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

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

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

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

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

¹⁸ Исходя из таблицы истинности

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

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

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

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

Логические выражения можно комбинировать:

```
>>> 1+3 > 7    # приоритет + выше, чем >
False
>>> (1+3) > 7 # скобки способствуют наглядности и избавляют от ошибок
False
>>> 1+(3>7)   # подумайте, что вернет выражение и почему
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.

Строки в Python тоже можно сравнивать по аналогии с числами. Начнем издалека. Символы, как и все остальное, представлено в компьютере в виде чисел. Есть специальная таблица, которая ставит в соответствие каждому символу некоторое число¹⁹.

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

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

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

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

Для сравнения строк Python их сравнивает посимвольно:

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

¹⁹ [Абсолютный Минимум, который Каждый Разработчик Программного Обеспечения Обязательно Должен Знать о Unicode и Наборах Символов](#)

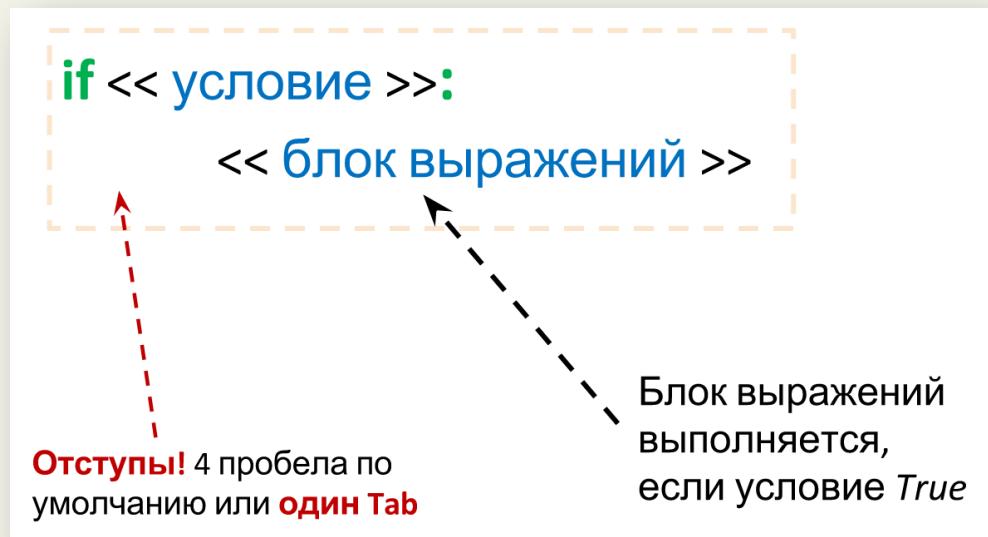
Следующий полезный оператор, с которым мы познакомимся – `in`. Он проверяет наличие подстроки в строке:

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

Освоив логические операции, перейдем к их использованию.

7. Условный оператор if

Наиболее часто логические выражения используются внутри условного оператора `if`:



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

Для примера обратимся к таблице с водородными показателями²⁰ 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

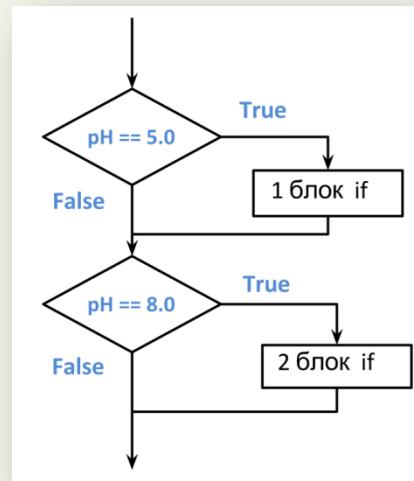
²⁰ [Википедия](#)

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

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

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

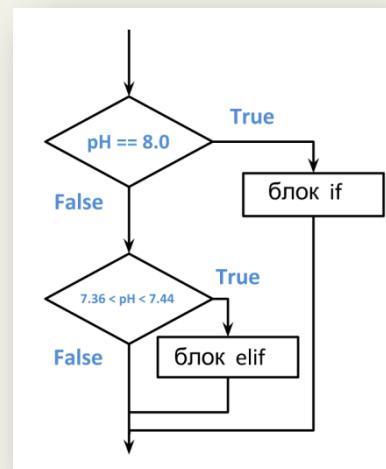
```
>>>
```



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

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

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



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



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

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.
>>>
```

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

Упражнение 7.1

Напишите программу, которая запрашивает у пользователя значение pH (с плавающей точкой) и выводит на экран вещество, соответствующее введенному pH²¹ (“Яблочный сок”, “Шампунь”, “Мыло для рук”). Определение pH производится в отдельной функции, которая возвращает строку с названием вещества или фразу “Не найдено”.

Упражнение 7.2

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

Упражнение 7.3

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

Упражнение 7.4

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

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

Упражнение 7.5

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

Упражнение 7.6

Напишите программу, которая в зависимости от характера ветра выдает сообщение о его скорости от 1 до 4 м/с – слабый; от 5-10 м/с – умеренный; от 9-18 м/с – сильный; больше 19 м/с – ураганный.

Упражнение 7.7

Напишите программу для расчета стоимости билетов на фильм, выбрав зал и сеанс. Ввести количество билетов и определить их стоимость с учетом, если заказывается более пяти билетов – скидка 5%, более 10 билетов – 10%. Красный зал – фильм «Пятница», сеансы 12 часов – 250 руб, 16 – 350 руб, 20 – 450 руб. Синий зал – фильм «Чемпионы: Быстрее. Выше. Сильнее», сеансы 10 часов – 250 руб, 13 – 350 руб, 16 – 450 руб. Голубой зал – фильм «Пернатая банда», сеансы 10 часов – 350 руб, 14 – 450 руб, 18 – 450 руб.

²¹ Смотрите значения в таблице водородных показателей: [Википедия](#)

8. 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:

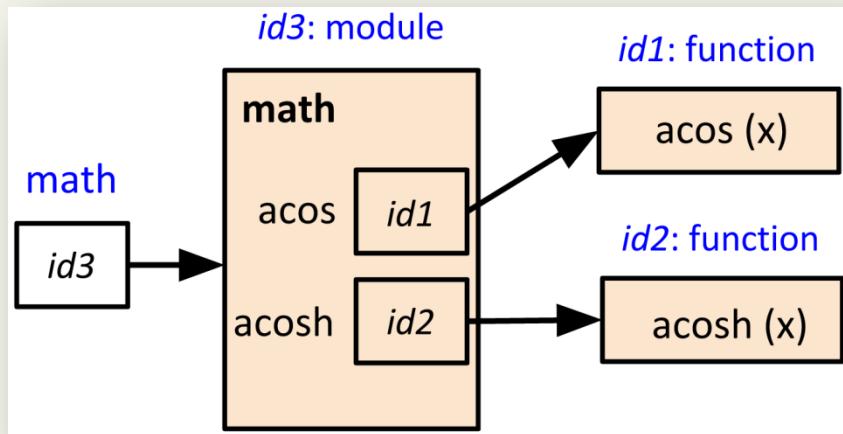
```
>>> help(math.sqrt)  
Help on built-in function sqrt in module math:  
  
sqrt(...)  
    sqrt(x)  
  
    Return the square root of x.  
>>>
```

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

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

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

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



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

В Python можно импортировать отдельную функцию из модуля:

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

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

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

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

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.

9. Создание собственных модулей

Теперь попробуем создать собственный модуль.

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

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

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

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

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

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

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

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

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

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

Новый модуль будет содержать вызов функции `print()`.

Импортируем его несколько раз подряд:

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

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

быть, если 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.

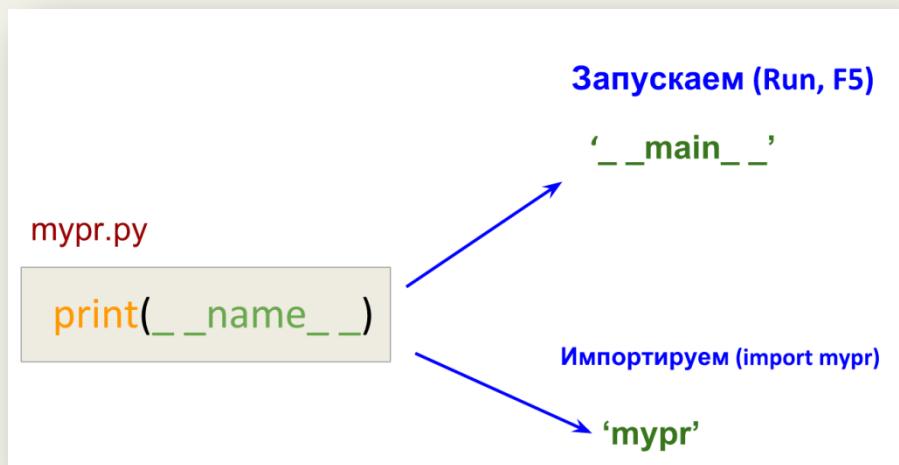
Продолжим эксперименты с модулями в Python. Создадим еще один модуль с именем `mypr.py`:

```
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):



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

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

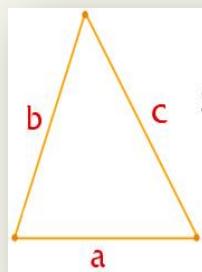
```
def func(x):
    return x**2+7

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

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

Упражнение 9.1

Найдите площадь треугольника с помощью формулы Герона. Стороны задаются с клавиатуры. Реализовать вычисление площади в виде функции, на вход которой подаются три числа, на выходе – площадь. Функция находится в отдельном модуле, где происходит разделение между запуском и импортированием. Описание математических функций можно найти в документации²²


$$S = \sqrt{p(p-a)(p-b)(p-c)}$$
$$p = \frac{a+b+c}{2}$$

Упражнение 9.2

Вывести число Пи с точностью до сотых.

Упражнение 9.3

Создайте в отдельном модуле функцию для вычисления выражения:

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

²² <https://docs.python.org/3/library/math.html>

Упражнение 9.4

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`²³.

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:

²³ <https://docs.python.org/3/library/random.html>

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

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

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

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

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

В результате выполнения программы получим:

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

Теперь вы умеете создавать собственные тесты!

Упражнение 9.5

Напишите функцию, вычисляющую значение:

$$x^4 + 4^x$$

Автоматизируйте процесс тестирования функции с помощью модуля `doctest`.

Для справки. Философия Python

Если импортировать модуль с именем `this`, то Python отобразит на экране свою философию²⁴:

```
>>> 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 9.6

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 9.7

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

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

²⁴ Перевод тут: <https://ru.wikipedia.org/wiki/Python>

Упражнение 9.8

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

10. Строковые методы в Python

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

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

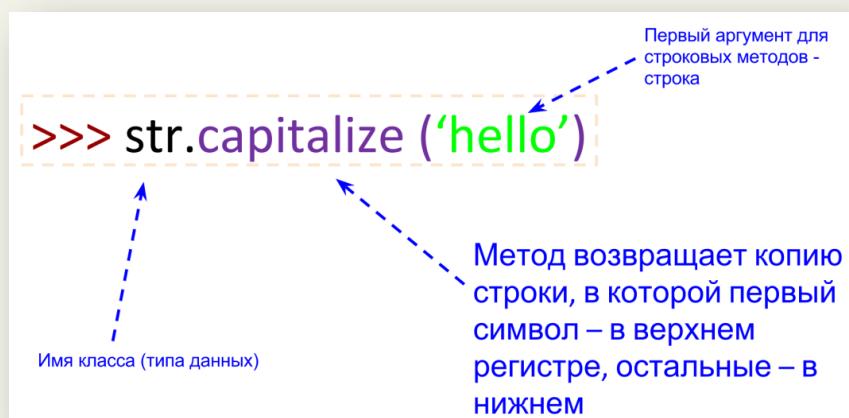
Функция сообщила нам, что объект 0 относится к классу '`int`', т.е. **тип данных является классом** (тип данных и класс – синонимы).

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

Рассмотрим пример вызова строкового метода:

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

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



Метод – это обычная функция, расположенная внутри класса.

Вызовем еще один метод:

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

Этот метод принимает два аргумента – строку и число:

```
>>> 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.

```
>>>
```

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

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

Не сложно догадаться, что делает метод `count()`.
Python содержит интересный метод `format()`²⁵:

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

Вместо `{0}` и `{1}` подставляются аргументы методы `format()`.
Поменяем их местами:

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

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

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

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

При работе с текстами полезно использовать строковый метод `strip()`:

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

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:

²⁵ <https://docs.python.org/3.1/library/string.html#format-examples>

```
>>> '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 методы²⁶:

`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

Для справки. Специальные строковые методы

Объединим две строки:

```
>>> 'ТТ' + 'rr'
'TTrr'
>>>
```

²⁶ Документация: <https://docs.python.org/3/library/stdtypes.html#string-methods>

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

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

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

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

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

Упражнение 10.1

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

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

Упражнение 10.2

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

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

11. Списки в Python

11.1. Создание списка

Начнем с примера. Предположим, что нам необходимо обработать информацию о курсах валют²⁷:

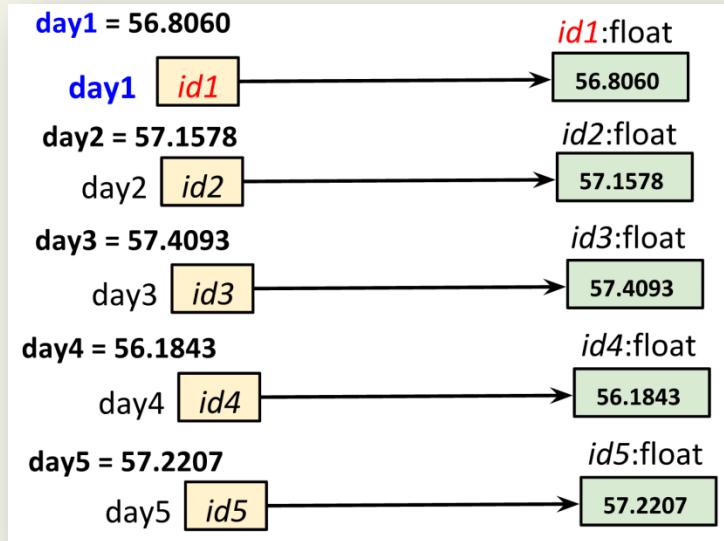
Дата	Доллар США USD	ЕВРО EUR
16.05.2015	50.0115	56.9881
15.05.2015	50.0774	57.1383
14.05.2015	49.5366	55.7138
13.05.2015	50.9140	57.1102
09.05.2015	50.7511	56.8971
08.05.2015	50.3615	57.2207
07.05.2015	49.9816	56.1843
06.05.2015	51.7574	57.4093
01.05.2015	51.1388	57.1578
30.04.2015	51.7029	56.8060
29.04.2015	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
>>>
```

²⁷ <http://www.sberometer.ru/cbr/>

Схематично:



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

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

Список (`list`) в Python является объектом²⁸, поэтому может быть присвоен переменной (переменная, как и в предыдущих случаях, хранит адрес объекта класса список).

Представим список для нашей задачи с курсом валют:

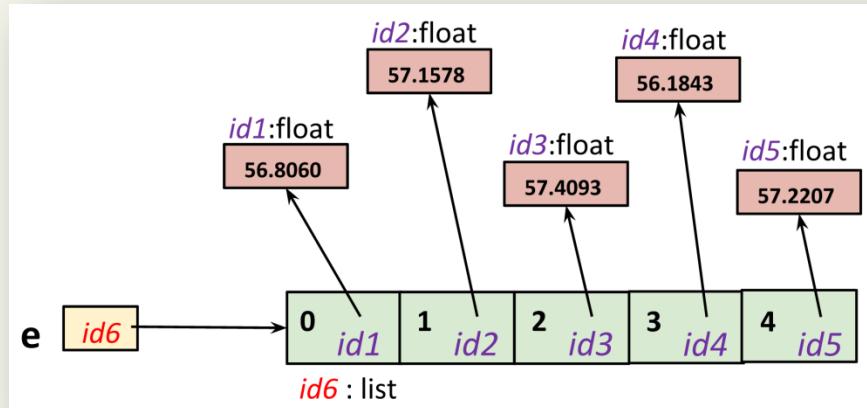
```

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

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

²⁸ В Python все являются объектами

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.

11.2. 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
>>>
```

Обращение по несуществующему индексу вызовет ошибку:

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

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

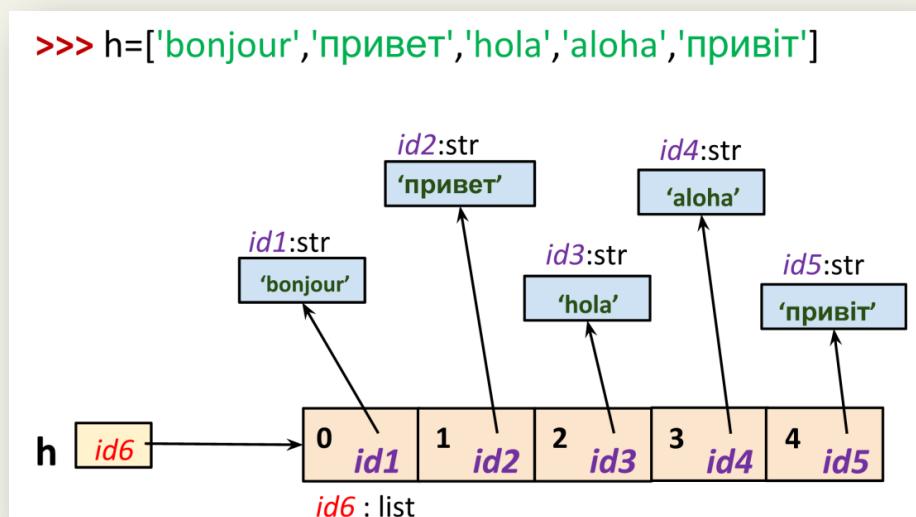
Списки можно изменить. Проведем эксперимент:

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

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

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

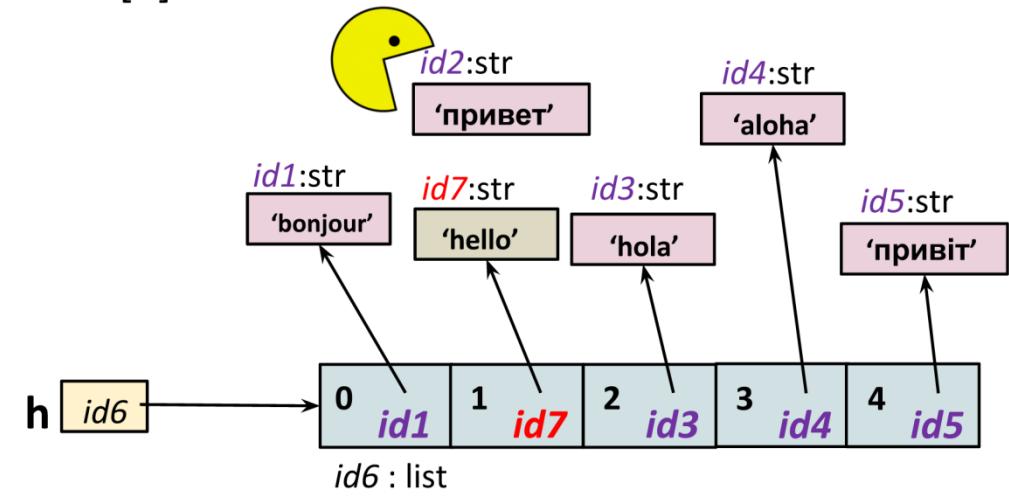
В памяти:



Производим изменения списка:

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

>>> h[1]='hello'



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]
>>>
```

Упражнение 11.1

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

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

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

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

Операция повторения (снова аналогия со строками):

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

Инструкция del позволяет удалять из списка элементы по индексу:

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

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

Получится следующее:

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

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

Теперь передадим в качестве аргументов две строки:

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

Передадим два числа:

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

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

Следующий полезный оператор `in` (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 11.2

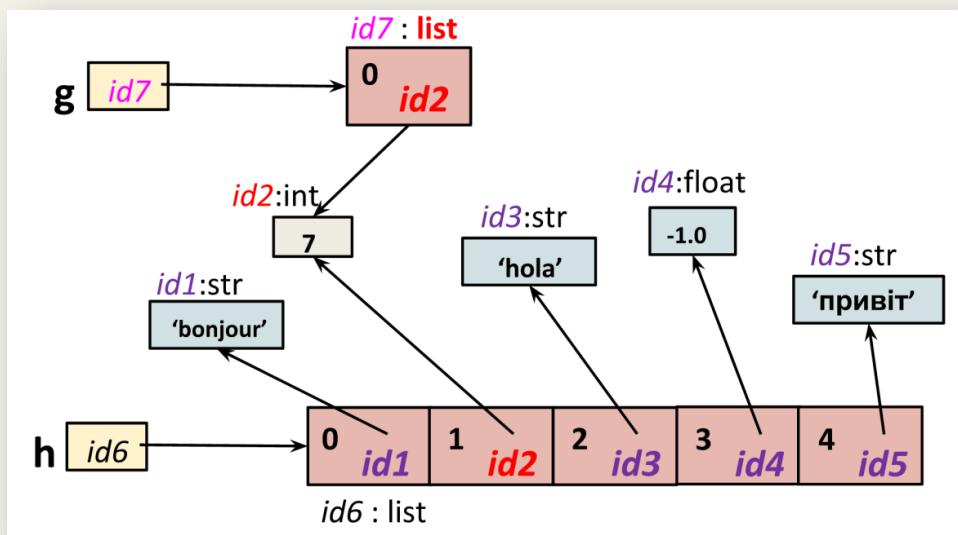
`L = [3, 'hello', 7, 4, 'привет', 4, 3, -1]`

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.

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

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

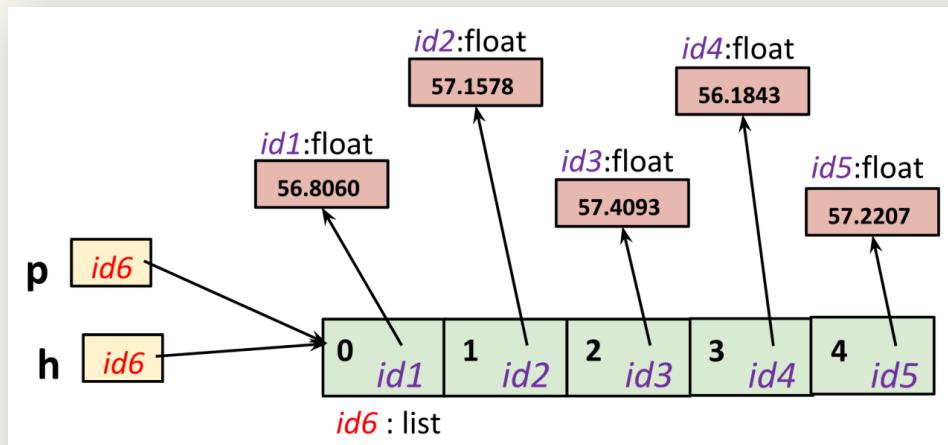
11.3. Псевдонимы и клонирование списков

Рассмотрим важную особенность списков. Выполним следующий код:

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

В Python две переменные называются псевдонимами²⁹, когда они содержат одинаковые адреса памяти.

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



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

²⁹ Псевдонимы – альтернативные имена чего-либо

Получить полную копию списка можно с помощью операции взятия среза [:] :

```
>>> h
['bonjour', 7, 'hola', -1.0, 'привіт']
>>> t=h[:] # теперь t - клон h (ссылается на копию)
>>> t
['bonjour', 7, 'hola', -1.0, 'привіт']
```

Теперь переменная t ссылается на копию списка h (клонировали список).

Аналогично можно создавать копии подсписков:

```
>>> y=h[:3]
>>> y
['bonjour', 7, 'hola']
```

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 объект
True
>>> x = [1, 2]
>>> y = [1, 2]
>>> x is y
False
>>>
```

С одной стороны список предоставляет возможность модификации, с другой – появляется опасность изменить псевдоним списка.

Упражнение 11.3

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

Исследуйте несколько примеров использования срезов (выполняются аналогично строкам).

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

11.4. Методы списка

Вернитесь к главе 10 и вспомните, что мы говорили о строковых методах. Для списков ситуация будет аналогичная.

Далее приведены наиболее популярные методы списка³⁰:

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

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.

³⁰ <https://docs.python.org/3/tutorial/datastructures.html#more-on-lists>

11.5. 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'
>>>
```

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

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

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

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

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

Возьмем другой разделитель:

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

Упражнение 11.4

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

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

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

11.6. Вложенные списки

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

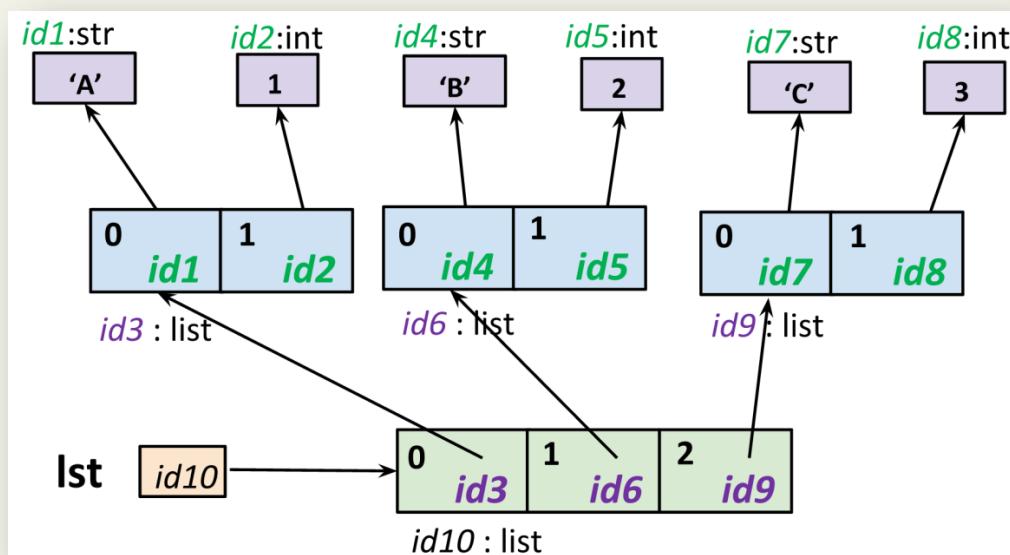
```
>>> lst=[['A', 1], ['B', 2], ['C', 3]]
>>> lst
[['A', 1], ['B', 2], ['C', 3]]
>>> lst[0]
['A', 1]
>>>
```

Подобные структуры используются для хранения матриц.

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

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

Схематично сложенные списки выглядят следующим образом:



Упражнение 11.5

Напишите программу, которая запрашивает у пользователя две строки и формирует из этих строк список. Если строки состоят только из чисел, то программа добавляет в середину списка сумму введенных чисел, иначе добавляется строка, образованная из слияния двух введенных ранее строк. Итоговая строка выводится на экран.

Упражнение 11.6

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.

12. Операторы цикла в Python

Язык Python позволяет быстро создавать прототипы³¹ реальных программ благодаря тому, что в него заложены конструкции для решения типовых задач, с которыми часто приходится сталкиваться программисту.

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

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

12.1. Оператор цикла *for*

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

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

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

³¹ Быстрая, черновая реализация будущей программы.

Перепишем этот пример с использованием цикла for:

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

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

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

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

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

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')  
  
7.0 - число 7.0  
>>>
```

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

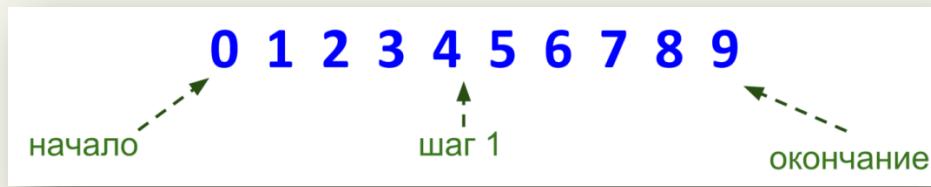
Похожим образом в цикле производится поиск необходимого символа в строке с помощью вызова строкового метода:

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

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

12.2. Функция `range()`

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



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

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

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
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!

В Python есть более красивое решение данной задачи:

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

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

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

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

Упражнение 12.1

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

Упражнение 12.2

`L = [-8, 8, 6.0, 5, 'строка', -3.1]`

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

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

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

В результате выполнения программы:

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

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

12.3. Создание списка

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:

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

В примере мы выбираем последовательно значения из списка `a`, возводим в квадрат каждый из его элементов и сразу добавляем полученные значения в новый список.
По аналогии можно перебирать символы из строки и формировать из них список:

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

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

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

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

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

Пример вызова `map()` для строки:

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

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

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

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

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

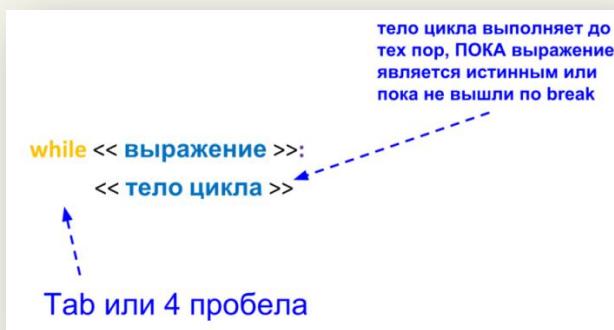
Упражнение 12.3

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

12.4. Оператор цикла `while`

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

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



Определим количество кроликов:

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

В результате выполнения программы:

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

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

Рассмотрим подробнее ход выполнения программы.

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

Рассмотрим следующий пример:

```
text = ""
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 12.4

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)
```

Результат выполнения:

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

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

Упражнение 12.5

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

Упражнение 12.6

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

Упражнение 12.7

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

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

Подсказка: вспомните про модификацию списков.

Упражнение 12.8

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

Упражнение 12.9

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

12.5. Вложенные циклы

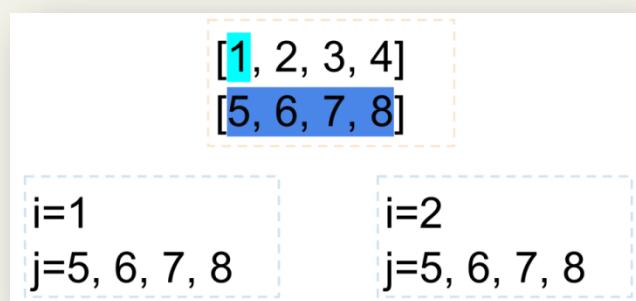
Циклы можно вкладывать друг в друга.

```
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 (см. п. 11.6).

Сначала пример с одним циклом `for`:

```
lst = [[1, 2, 3],  
       [4, 5, 6]]  
  
for i in lst:  
    print (i)
```

Результат выполнения программы:

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

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

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

```
lst = [[1, 2, 3],  
       [4, 5, 6]]  
  
for i in lst:      # цикл по элементам внешнего списка  
    print()  
    for j in i:    # цикл по элементам элементов внешнего списка  
        print (j, end="")
```

Результат выполнения программы:

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

Упражнение 12.10

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

Упражнение 12.11

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

Упражнение 12.12

Дана матрица (см. упражнение 12.10). Вывести номер строки, содержащей максимальное число одинаковых элементов.

Упражнение 12.13

Дана целочисленная квадратная матрица (см. упражнение 12.11). Найти произведение элементов матрицы, лежащих ниже главной диагонали.

Упражнение 12.14

Дана целочисленная квадратная матрица (см. упражнение 12.11). Найти сумму элементов матрицы, лежащих выше главной диагонали.

На заметку

Операторы 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
>>>
```

13. Множества

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

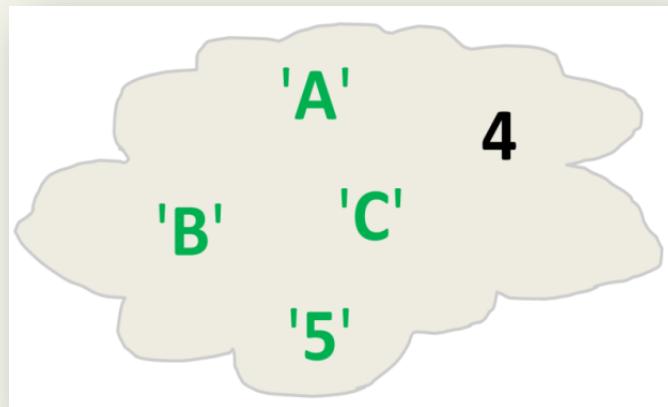
Рассмотрим множество (`set`) в Python – неупорядоченную коллекцию неизменяемых, уникальных элементов.

Создадим множество:

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

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

Представим множество схематично:



Множества в Python обладают интересными свойствами:

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

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

Рассмотрим способы создания множеств:

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

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

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

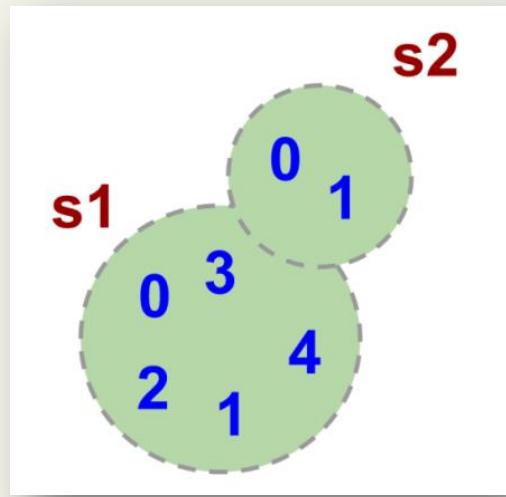
Функция `range()` позволяет создавать множества из диапазона:

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

Рассмотрим некоторые операции над множествами³²:

```
>>> 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'}
>>>
```

³² <https://docs.python.org/3/tutorial/datastructures.html#sets>

14. Кортежи

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

Некоторые операции над кортежами³³:

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

С помощью кортежей можно присваивать значения одновременно двум переменным:

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

Поменять местами содержимое двух переменных:

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

³³ <https://docs.python.org/3/tutorial/datastructures.html#tuples-and-sequences>

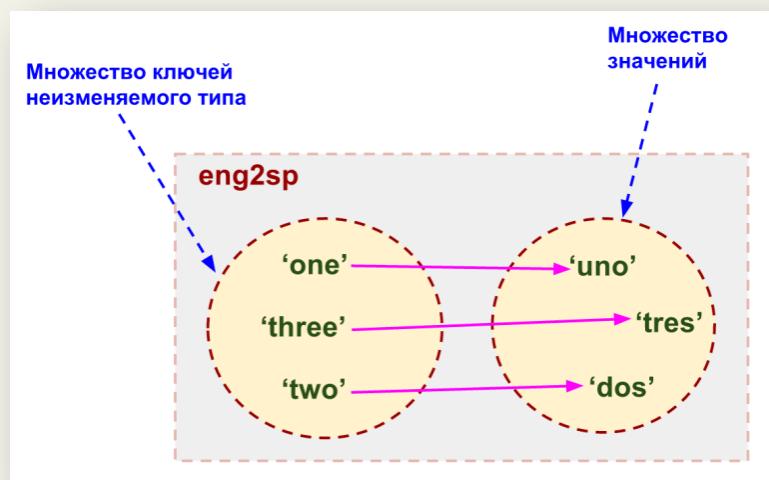
Мы сказали, что кортеж нельзя изменить, но можно изменить, например, список, входящий в кортеж:

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

15. 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'}
>>>
```

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

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

Результирующий словарь eng2sp отобразился в «перемешанном» виде, т.к., по аналогии с множествами, словари являются неупорядоченной коллекцией. Фактически, словарь – это отображение двух множеств: множества ключей и множества значений.

К словарям применим оператор `in`:

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

Часто словари используются, если требуется найти частоту встречаемости элементов в **последовательности (списке, строке, кортеже³⁴)**.

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

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

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

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

³⁴ Вы, наверно, обратили внимание, что все эти типы данных имеют общие свойства, поэтому их относят к последовательностям. [Документация](#)

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'
>>>
```

16. Несколько слов об алгоритмах

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

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

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

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

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

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

Какие возможны алгоритмы решения?

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

Рассмотрим каждый из перечисленных алгоритмов.

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

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

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

[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 отсортованного 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.

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

На первом шаге просматриваем первые два элемента списка:

[809, 834, ...]
индекс: 0 1

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

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

809	834
min1	min2

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

Просматриваем следующий элемент списка (477):

809	834	[..., 477, ...]
min1	min2	

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

477	809	834
min1	min2	

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

477	809
min1	min2

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)
```

Специально не останавливался на теории построения и оценки алгоритмов³⁵. Приведу книги, где об этом говорится хорошо и подробно:

1. Томас Х. Кормен. *Алгоритмы. Вводный курс*³⁶
2. Томас Х. Кормен. *Алгоритмы. Построение и анализ*³⁷
3. Стивен С. Скиена. *Алгоритмы. Руководство по разработке*³⁸

Упражнение 16.1

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

Упражнение 16.2

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

В упражнениях 16.3-8 список состоит из случайных элементов, в списке не менее 1000 элементов.

Упражнение 16.3

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

Упражнение 16.4

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

Упражнение 16.5

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

Упражнение 16.6

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

Упражнение 16.7

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

Упражнение 16.8

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

Упражнение 16.9

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

³⁵ Стандартные функции (методы) в Python являются наиболее быстрыми.

³⁶ [Книга для самых начинающих](#)

³⁷ [Более подробно об алгоритмах](#)

³⁸ <http://www.ozon.ru/context/detail/id/6290126/>

17. 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 предлагает³⁹ другой способ, основанный на перехвате ошибок (исключений).

³⁹ Другие объектно-ориентированные языки тоже поддерживают данный механизм

Перепишем наш пример с учетом возможностей Python:

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

Выполним программу:

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

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

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

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

Перейдем в интерактивный режим Python и выполним несколько команд, приводящих к ошибкам (исключениям⁴⁰):

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

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

⁴⁰ [Типы исключений в Python](#)

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!  
>>>
```

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

Перехват исключений используется при написании функций, например:

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

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

Результат выполнения:

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

Упражнение 17.1

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

18. 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.

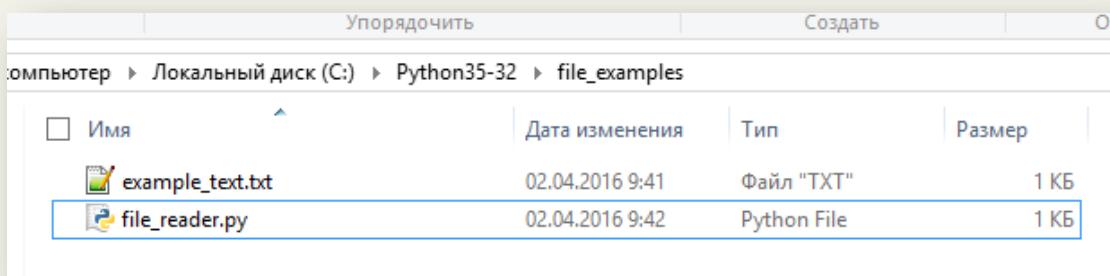
1. Создайте каталог (папку) `file_examples`.
2. С помощью (например, Блокнота) создайте в каталоге `file_examples` текстовый файл `example_text.txt`, содержащий следующий текст:

```
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()
```

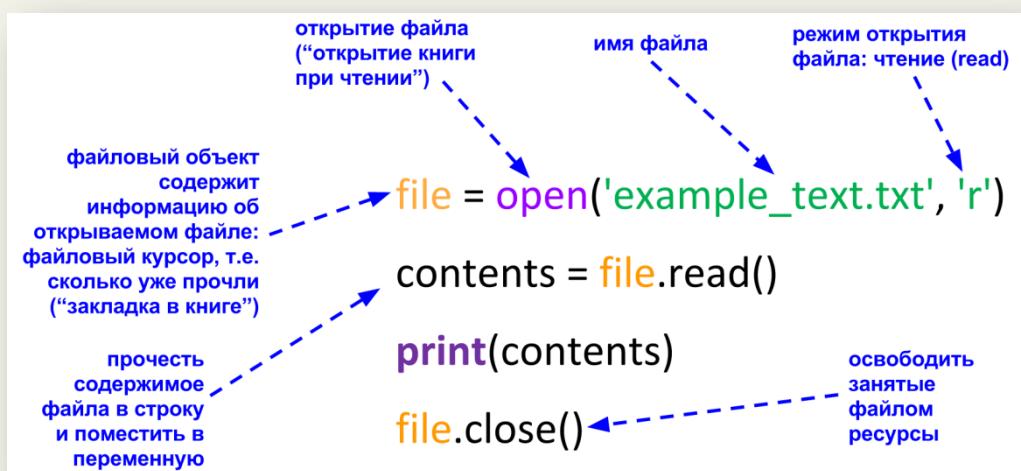
Получится примерно следующее:



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
>>>
```

Небольшие комментарии к исходному тексту:



Рассмотренный подход по работе с файлами⁴¹ в Python перешел из языка С.

По умолчанию, если не указывать режим открытия, то используется открытие на «чтение».

Файлы особенно подвержены ошибкам во время работы с ними. Диск может заполниться, пользователь может удалить используемый файл во время записи, файл могут переместить и т.д. Эти типы ошибок можно перехватить с помощью обработки исключений:

⁴¹ [Документация для работы с файлами в Python](#)

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

Запустим программу:

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

В дальнейшем для работы с файлами мы будем использовать *менеджер контекста* (инструкцию `with`⁴²), который не требует ручного освобождения ресурсов.

Перепишем предыдущий пример с использованием менеджера контекста:

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

Выполним программу:

```
>>>
== 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'
>>>
```

⁴² Менеджер контекста используется не только при работе с файлами

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)
```

Результат выполнения:

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

Следующий пример демонстрирует работу с курсором:

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

Результат работы программы:

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

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

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

Результат работы программы:

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

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

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

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

```
with open('plan.txt', 'r') as file:
    planets = file.readlines()

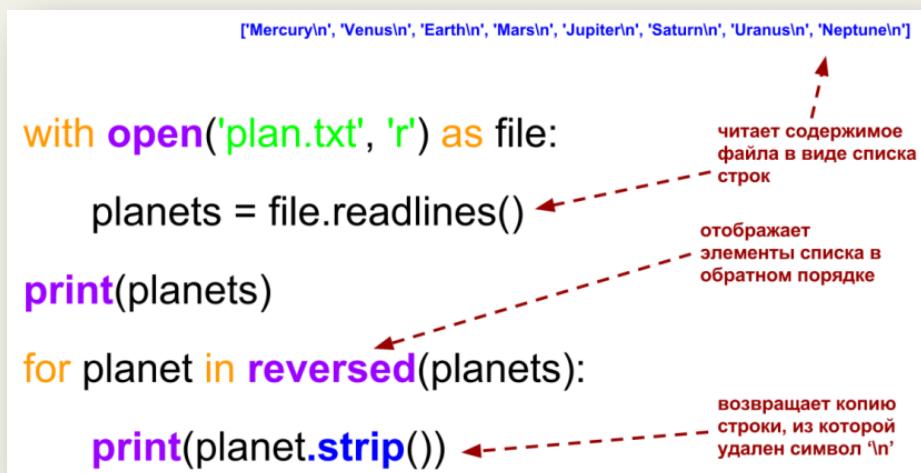
print(planets)

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()))
```

Результат выполнения:

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

7
Venus

5
Earth

5
Mars

4
Jupiter

7
Saturn

6
Uranus

6
Neptune
7
>>>
```

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

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

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

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

Упражнение 18.1

Отсортированное по алфавиту содержимое файла *plan.txt* поместите в файл *sort_plan.txt*.

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)
```

Упражнение 18.2

Напишите программу, которая создает (генерирует) полноценный HTML-документ, содержащий текст, приведенный по ссылке: <http://dfedorov.spb.ru/python/files/tutchev.txt> и под текстом размещает картинку: <http://dfedorov.spb.ru/python/files/tutchev.jpg>. Пример итогового HTML-документа: <http://dfedorov.spb.ru/python/files/p.html> (обратите внимание на код страницы, содержащей HTML-теги).

Выполните обработку ошибок.

PS. в момент чтения и записи используйте параметр функции open() encoding='utf-8'.

Для справки. Регулярные выражения

Python поддерживает мощный язык регулярных выражений⁴³, 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 match⁴⁴, иначе возвращается None.

Упражнение 18.3

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 18.4

Очистите файл от HTML-тегов: <http://dfedorov.spb.ru/python/files/p.html>

Выведите на экран «чистый» текст. PS. можно использовать только стандартные модули Python.

⁴³ [Документация по регулярным выражениям](#)

⁴⁴ [Документация по объекту match](#)

Упражнение 18.5

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

Упражнение 18.6

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

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

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

Поля (столбцы) файла:

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

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

Кодировка файла: 'cp1251'

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

Сортировка списка:

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

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

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

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

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']
```

19. Объектно-ориентированное программирование в Python

19.1 Основы объектно-ориентированного подхода

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

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

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

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

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

Вызов функции, передача аргументов:

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

В результате работы программы:

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

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

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

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)⁴⁵ класса Address⁴⁶:

```
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"
```

⁴⁵ В Python классы являются объектами, но для упрощения скажем, что это шаблон

⁴⁶ Вспомните о создании объекта класса int: a = int ()

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

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

Зададим поля объекта, адрес которого находится в переменной vacationHomeAddress:

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

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

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

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

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

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

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

В результате выполнения программы получим:

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

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

Создадим класс Dog:

```
class Dog():
    age=0      # возраст собаки
    name=""    # имя собаки
    weight=0   # вес собаки
    # Первым аргументом любого метода всегда является self, т.е. сам объект
    def bark(self): # функция внутри класса называется методом
        # 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 по средствам 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 19.1

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!
>>>
```

Рассмотрим пример присвоения имени собаки через вызов конструктора класса:

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

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

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

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

Результат работы программы:

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

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

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

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

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

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

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

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

Проверим, что все работает:

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

Упражнение 19.2

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 19.3

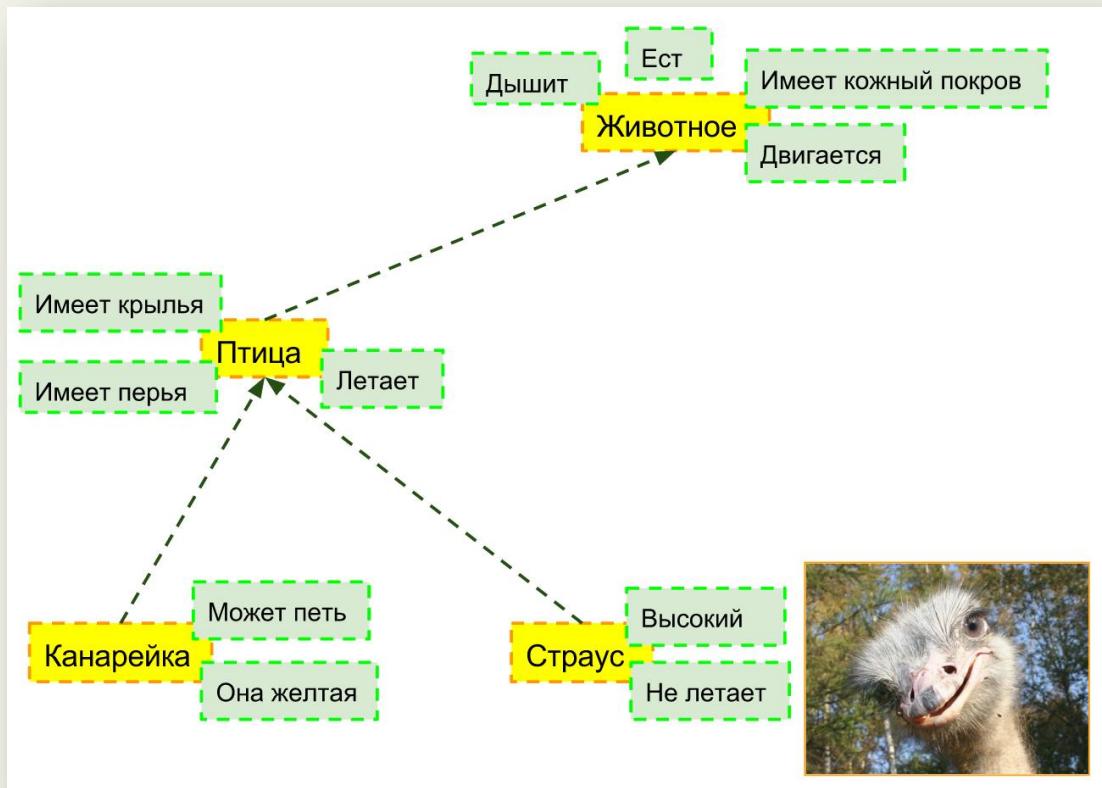
Создайте класс `StringVar` для работы со строковым типом данных, содержащий методы `set()` и `get()`. Метод `set()` служит для изменения содержимого строки, `get()` – для получения содержимого строки. Создайте объект типа `StringVar` и протестируйте его методы.

Упражнение 19.4

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

19.2. Наследование в Python

Объектно-ориентированный подход в программировании тесно связан с мышлением человека, с работой его памяти. Для того чтобы нам лучше понять свойства ООП, рассмотрим модель хранения и извлечения информации из памяти человека (модель предложена учеными Коллинзом и Квиллианом)⁴⁷. В своем эксперименте они использовали семантическую сеть, в которой были представлены знания о канарейке:



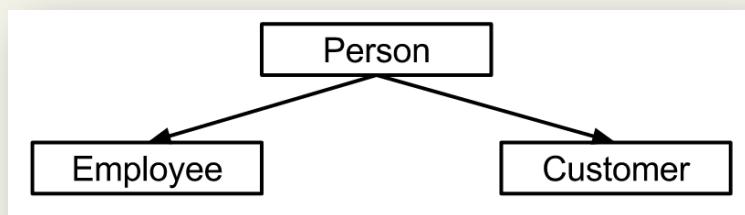
Например, «канарейка — это желтая птица, которая умеет петь», «птицы имеют перья и крылья, умеют летать» и т. п. Знания в этой сети представлены на различных уровнях: на нижнем уровне располагаются более частные знания, а на верхних — более общие. При таком подходе для понимания высказывания «Канарейка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

⁴⁷ Гаврилова Т.А., Муромцев Д.И. Интеллектуальные технологии в менеджменте: инструменты и системы

требует большего времени, т. к. человеку необходимо проследовать по иерархии понятий в своей памяти. С другой стороны, конкретные свойства могут перекрывать более общие, что также требует меньшего времени на обработку информации. Например, вопрос «Может ли страус летать» 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.

Помимо полей базового класса происходит наследование методов:

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

class Employee(Person):
    job_title=""

class Customer(Person):
    email=""

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

Результат работы программы:

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

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

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

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

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

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

Результат работы программы:

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

Видим, что в момент создания объекта вызывается конструктор, содержащийся в дочернем классе, т.е. конструктор дочернего класса переопределил конструктор базового класса.

Порой требуется вызвать конструктор базового класса из конструктора дочернего класса:

```
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

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 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,зывающий eat() и makeNoise() для каждого животного.

19.3. Иерархия наследования в Python

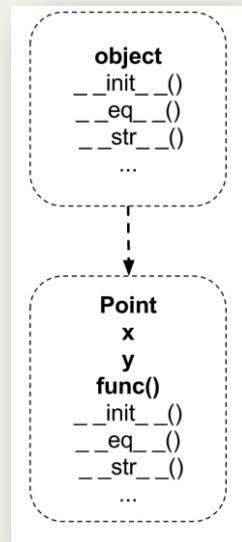
В Python все создаваемые классы наследуются от класса `object`. Создадим класс (собственный тип данных) `Point`, в котором определим (переопределем методы базового класса `object`) специальные методы `__init__()`, `__eq__()`, `__str__()`:

```
class Point:  
    def __init__(self, x=0, y=0): # конструктор устанавливает координаты  
        self.x = x  
        self.y = y  
    def __eq__(self, other): # метод для сравнения двух точек  
        return self.x == other.x and self.y == other.y  
    def __str__(self): # метод для строкового вывода информации  
        return "({0.x}, {0.y})".format(self)  
    def func(self): # понадобится в следующем примере  
        return abs(self.x-self.y)  
  
a = Point() # создаем объект, по умолчанию x=0, y=0  
print(str(a)) # здесь вызывается метод __str__() класса Point  
# полная форма Point.__str__(a)  
b = Point(3, 4)  
print(str(b))  
b.x = -19  
print(a.func())  
print(str(b))  
print(a == b, a != b) # вызывается метод __eq__()  
# полная форма для сравнения a == b имеет вид: Point.__eq__(a, b)
```

Результат работы программы:

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

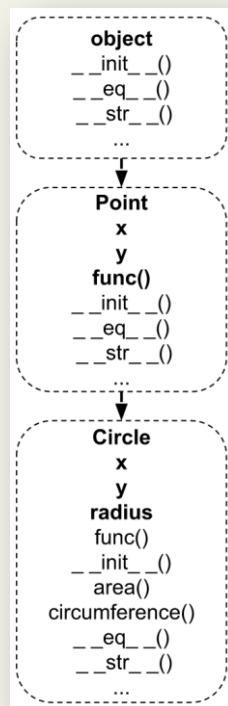
Схематично иерархия классов имеет следующий вид:



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.)⁴⁸.

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`:



⁴⁸ Подробнее: <https://docs.python.org/3/reference/datamodel.html>

Исходный код класса Circle:

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

Результат работы программы:

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

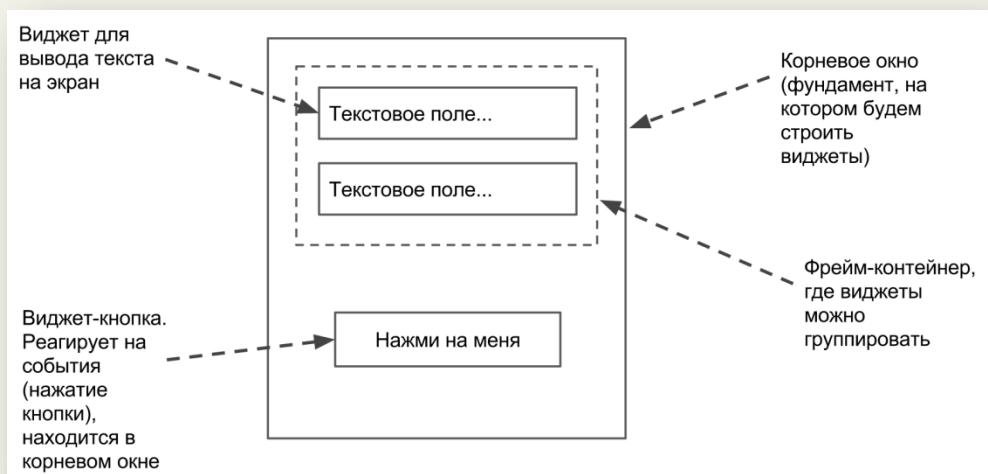
20. Разработка приложений с графическим интерфейсом

20.1. Основы работы с модулем *tkinter*

Язык Python позволяет создавать приложения с графическим интерфейсом, для этого используются различные графические библиотеки⁴⁹. Остановимся на рассмотрении стандартной (входит в стандартный комплект Python) графической библиотеки *tkinter*⁵⁰.

Первым делом при работе с *tkinter* необходимо создать главное (корневое) окно, в котором размещаются остальные графические элементы – виджеты. Существуют различные виджеты⁵¹ 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:

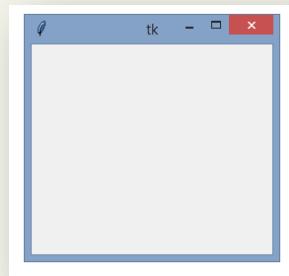
```
# Подключаем модуль, содержащий методы для работы с графикой
import tkinter
# Создаем главное (корневое) окно,
# в переменную window записываем ссылку на объект класса Tk
window = tkinter.Tk()
# Задаем обработчик событий для корневого окна
window.mainloop()
```

⁴⁹ [Список графических библиотек, поддерживаемых Python](#)

⁵⁰ [Подробнее про модуль tkinter](#)

⁵¹ [Перечень виджетов](#)

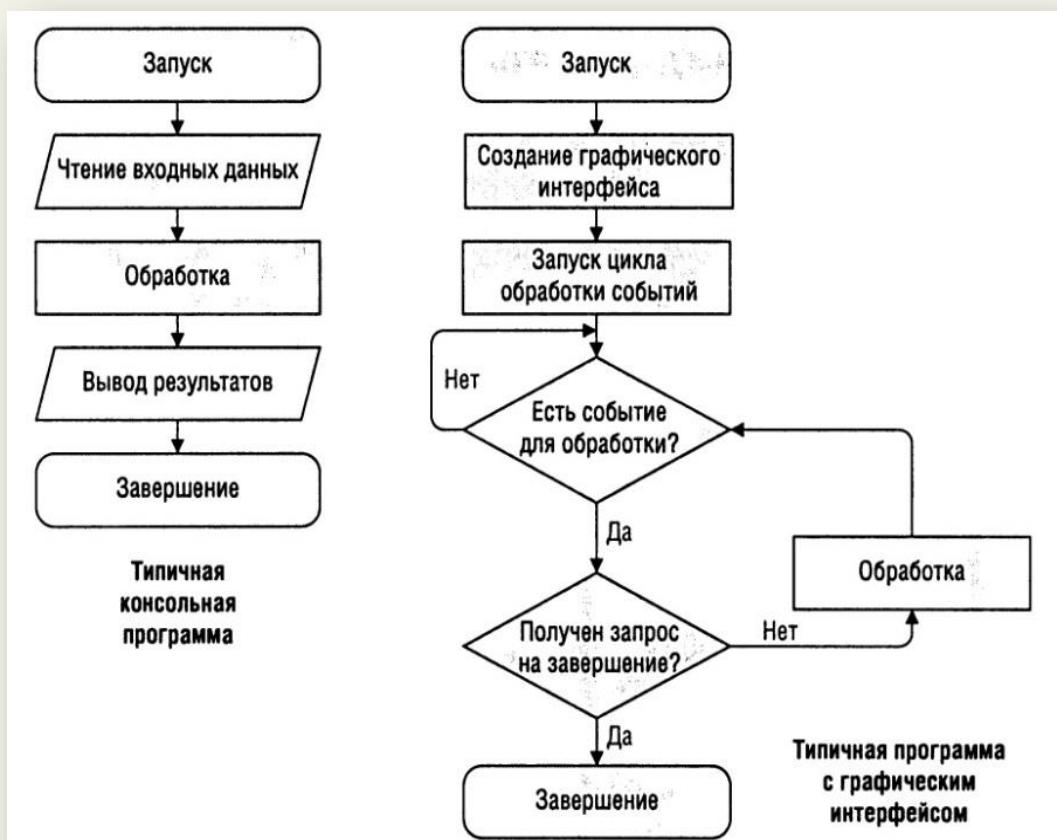
Результат выполнения программы:



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

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

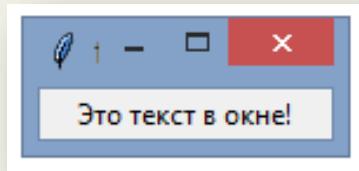
Слева на схеме показан алгоритм работы консольной программы, справа – программы с графическим интерфейсом:



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()
```

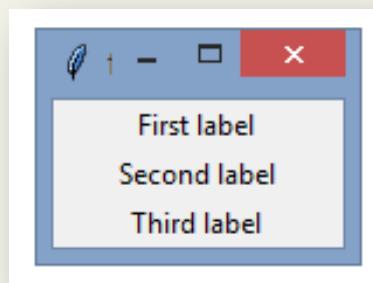
Результат работы программы:



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:



Можно изменять параметры фрейма в момент создания объекта⁵²:

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



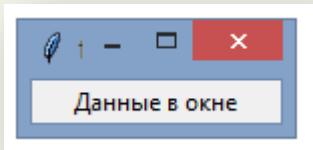
В следующем примере для отображения в виджете Label содержимого переменной, используется переменная data класса StringVar (из модуля tkinter). В дальнейшем из примеров станет понятнее, почему в tkinter используются переменные собственного класса⁵³.

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

⁵² <http://effbot.org/tkinterbook/frame.htm>

⁵³ Tkinter поддерживает работу с переменными классов: BooleanVar, DoubleVar, IntVar, StringVar

Результат выполнения программы:

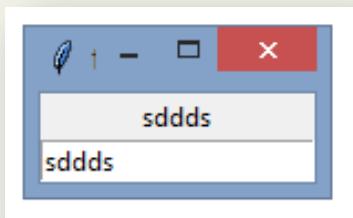


20.2. Шаблон «Модель-вид-контроллер» на примере модуля *tkinter*

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:

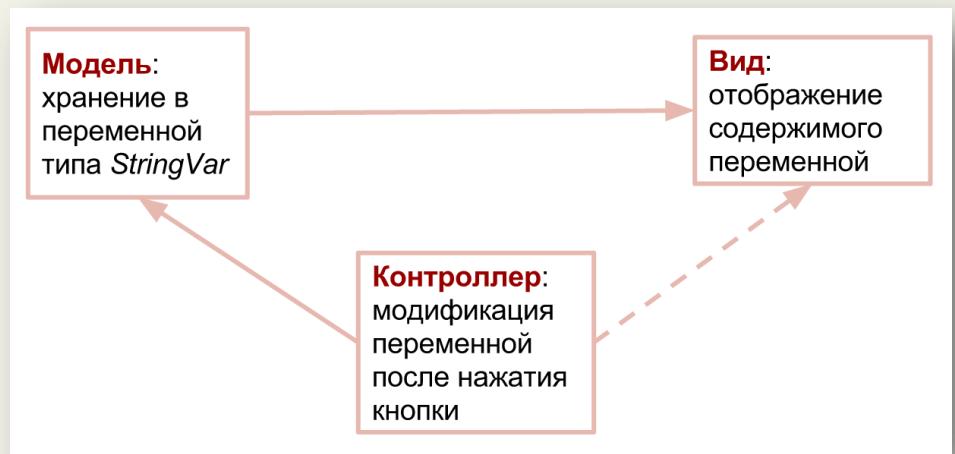


Видим, что текст, который мы набираем, мгновенно отображается в окне. Дело в том, что виджеты Label и Entry используют для вывода и ввода текста соответственно одну и ту же переменную data класса StringVar. Подобная схема работы оконного приложения укладывается в универсальный шаблон (паттерн), который называется «Модель-вид-контроллер» (Model-View-Controller или MVC)⁵⁴.

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.

Следующая схема показывает связь всех компонентов модели MVC:

⁵⁴ Паттерн MVC получил широкое распространение при разработке веб-приложений



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

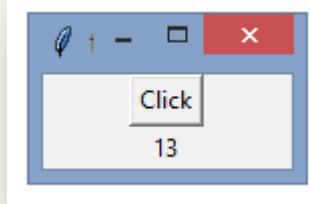
Следующий пример демонстрирует возможности обработки событий при нажатии на кнопку (виджет *Button*):

```
import tkinter

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

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

Результат выполнения программы:



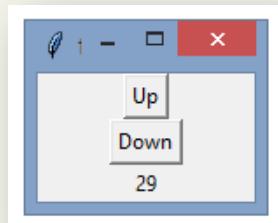
Более сложный пример с двумя кнопками и двумя обработчиками событий (`click_up`, `click_down`):

```
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:

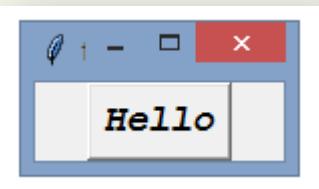


20.3. Изменение параметров по умолчанию при работе с tkinter

Tkinter позволяет изменять параметры виджетов в момент их создания:

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

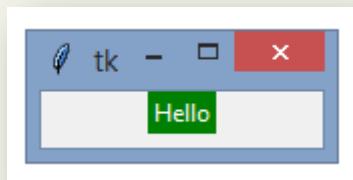
Результат выполнения программы:



В следующем примере изменяются параметры виджета Label:

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

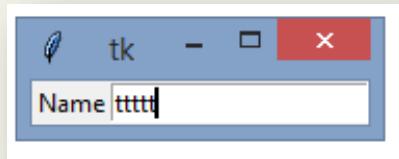
Результат выполнения программы:



Менеджер расположения (геометрии) pack тоже имеет параметры:

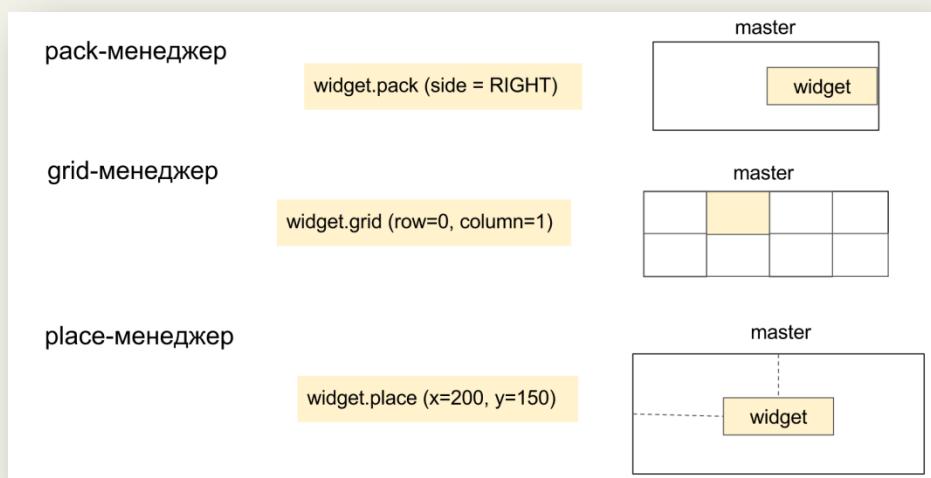
```
import tkinter
window = tkinter.Tk()
frame = tkinter.Frame (window)
frame.pack()
label = tkinter.Label (frame, text='Name')
# Выравнивание по левому краю
label.pack (side='left')
entry = tkinter.Entry (frame)
entry.pack (side='left')
window.mainloop()
```

Результат выполнения программы:



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:



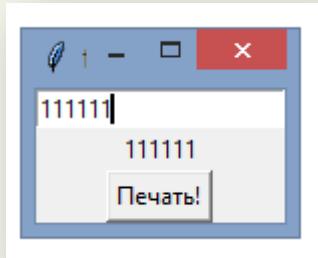
Особенность следующего примера в том, что введенный текст (через виджет `Entry`) отображается на экране (через виджет `Label`) только в момент нажатия кнопки (виджет `Button`), а не в реальном времени, как это было раньше:

```
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()
```

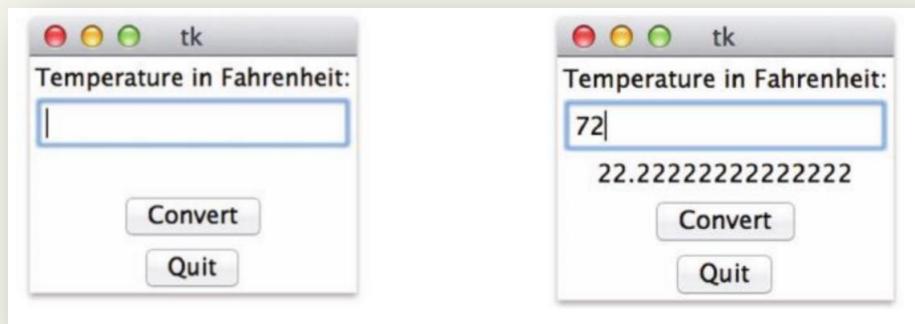
```
window.mainloop()
```

Результат выполнения программы:



Упражнение 20.1

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



Упражнение 20.2

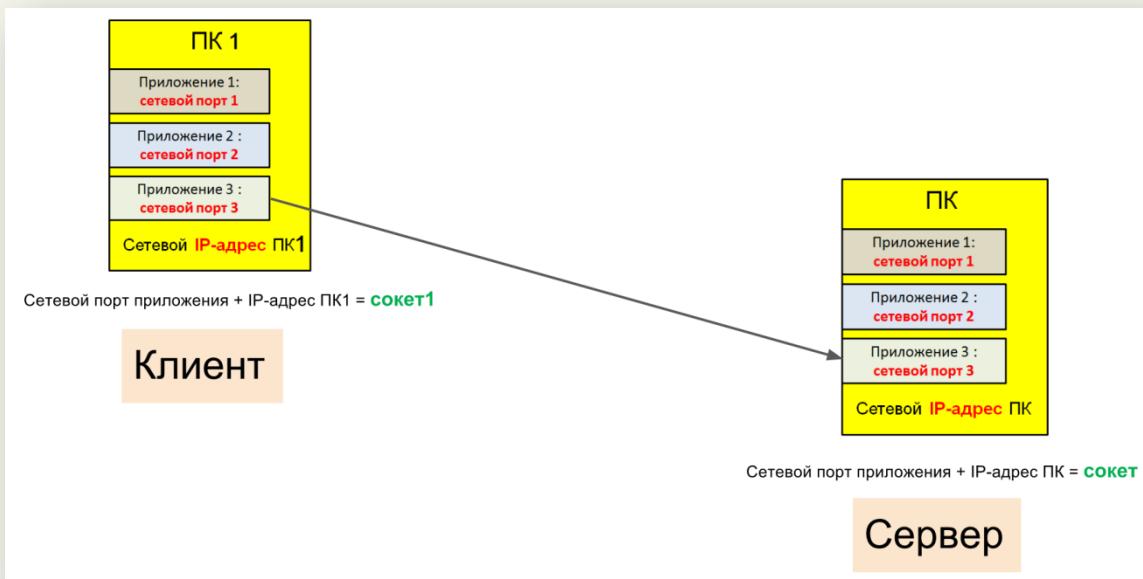
Напишите программу, которая отображает случайное слово на русском языке (тип данных `dict`). Пользователь пытается угадать его на английском (или другом языке).

PS. Можно ограничить работу программы по числу правильно угаданных слов (вести персональный рейтинг).

21. Клиент-серверное программирование в Python

Рекомендую небольшое видео о том, что такое сеть Интернет и TCP/IP за 15 минут:
<https://www.youtube.com/watch?v=v-rIM2YvTfA>

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



Для идентификации ПК в сети применяются IP-адреса⁵⁵, например, 192.168.0.3. На ПК работает большое число сетевых приложений (Skype, Telegram и пр.), поэтому, чтобы ПК определить, для какого приложения поступили данные, необходимо каждому сетевому приложению присвоить уникальный номер – сетевой порт⁵⁶ (например, Skype использует 80 и 443 порты). Связка «IP-адрес, сетевой порт» называется *сокетом* (*socket*). Сокеты предоставляют программный интерфейс для сетевого взаимодействия. Впервые они были реализованы на языке Си в системе BSD. Python имеет встроенный модуль *socket*^{57 58}.

Сетевое взаимодействие происходит по средствам клиент-серверного⁵⁹ обмена данными, где клиент – запрашивает (отправляет) данные, сервер – обрабатывает данные, полученные от клиента. Например, веб-клиентом является браузер, а веб-сервером – удаленный ПК, способный обрабатывать HTTP-запросы, поступающие от браузера.

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

⁵⁵ [Про IP-адрес](#)

⁵⁶ [Список портов](#)

⁵⁷ <https://docs.python.org/3.5/howto/sockets.html>

⁵⁸ <https://docs.python.org/3/library/socket.html>

⁵⁹ [Клиент-серверная архитектура](#)

Клиент-серверное взаимодействие в нашем примере будет происходить на одном и том же⁶⁰ ПК, поэтому в качестве IP-адреса указываем 127.0.0.1.

Сервер (обрабатывает поступающие запросы от клиента, листинг на [github](#)):

```
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, листинг на [github](#)):

```
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)
```

⁶⁰ <https://ru.wikipedia.org/wiki/Localhost>

Результат работы программы на стороне сервера:

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

Результат работы программы на стороне клиента:

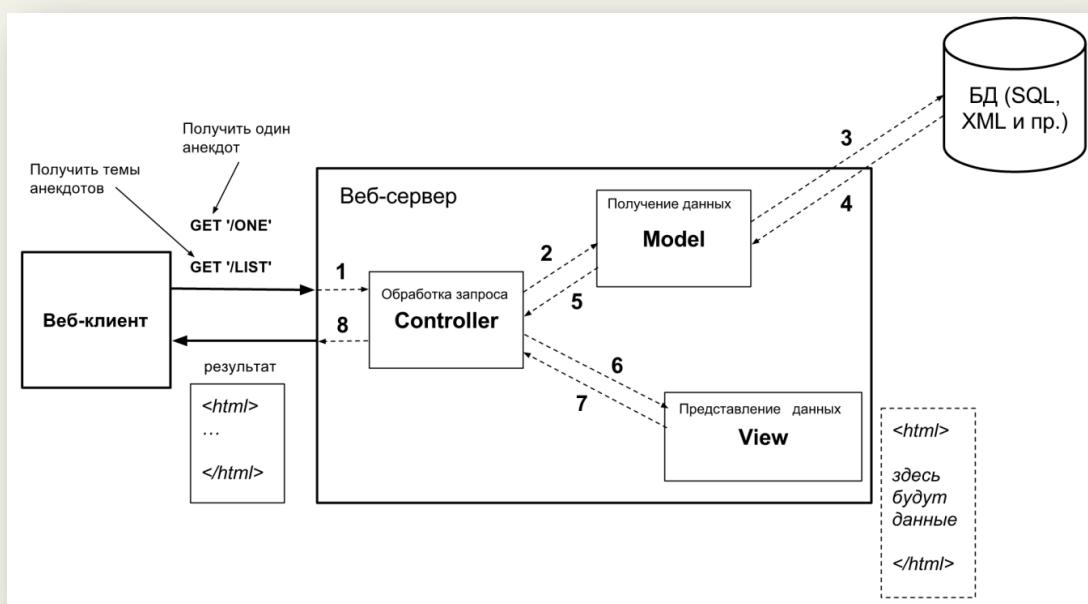
```
>>>
=====
RESTART: C:\Python35-32\client.py =====
Send[1]: Hello world
Received[4]: b'Hello world'
>>>
```

Упражнение 21.1

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

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

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

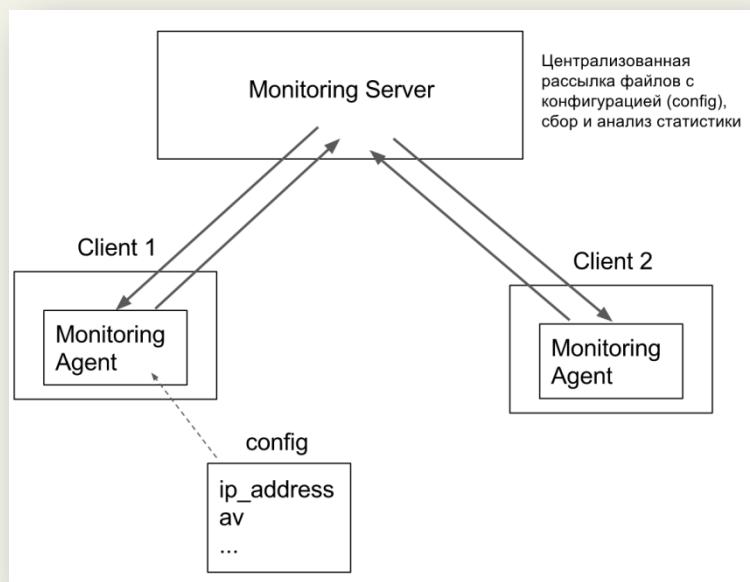


- 1 - отправка команды от клиента, команда попадает **Контроллеру** – проверка корректности команды и ее обработка;
- 2, 5 - запрос и получение данных от **Модели**;
- 3, 4 - запрос и получение данных из БД, файла и пр.;
- 6, 7 – «сырые» данные отправляются **Виду**, возвращаются данные, имеющие представление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

Предположим, что реализуется бот-анекдотов. Клиент (тонкий клиент⁶¹) 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 21.2

Разработать 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). На хосте производится логирование 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

⁶¹ [Тонкий клиент Википедия](#)

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 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).

Упражнение 21.3

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

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

Вопросы к зачету по языку программирования Python (базовый уровень)

1. История и тенденции развития языков программирования
2. Области применения языка программирования Python
3. Переменные в Python. Наименование. Модель памяти Python при работе с переменными
4. Функции в Python. Создание функций
5. Создание программ на языке Python в отдельном файле. Отличие от интерактивного режима
6. Строки и операции над строками в языке Python
7. Операторы отношений в Python. Логические операции над объектами
8. Условный оператор if
9. Модули в Python
10. Создание собственных модулей в Python
11. Строковые методы в Python. Отличие функций от методов
12. Списки в Python. Создание списка
13. Операции над списками в Python
14. Псевдонимы и клонирование списков в Python
15. Методы списка в Python
16. Преобразование типов в Python (списки, строки)
17. Вложенные списки в Python
18. Циклы в Python
19. Цикл for для списков и строк в Python
20. Функция range() и цикл for в Python
21. Способы генерации списка в Python
22. Цикл while в Python
23. Вложенные циклы в Python (на примере вложенных списков)
24. Множества и операции над ними в Python
25. Кортежи и операции над ними в Python
26. Словари и операции над ними в Python
27. Обработка исключений в Python
28. Работа с файлами в Python. Менеджер контекста
29. Объектно-ориентированное программирование в Python. Классы, объекты
30. Иерархия наследования в Python (класс object)
31. Полиморфизм в Python
32. Структура оконного приложения на примере модуля tkinter (обработка событий)
33. Реализация шаблона «Модель-вид-контроллер» на примере модуля tkinter
34. Структура клиент-серверного приложения (модуль socket)

22. Продвинутый уровень Python

22.1. Функциональный подход

в процессе

22.2. Импортирование модулей, написанных на языке C (для Python 3)

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

Для создания модулей на языке С воспользуемся пакетом `distutils`⁶², входящим в состав стандартной библиотеки Python.

Рассмотрим пример⁶³ создания собственного модуля на языке С. Для этого нам понадобится создать файл на языке С (`ownmod.c`), представляющий наш модуль:

```
#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() {
```

⁶² Подробнее: <https://docs.python.org/3.6/library/distutils.html>

⁶³ Источник: https://www.ibm.com/developerworks/ru/library/l-python_details_07/

```
// В Python 2 обходились без создания ownmodule:  
//(void) Py_Initialize( "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]  
)
```

Выполняем в командной строке (устанавливаем модуль⁶⁴):

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

Выполняем в командной строке с правами администратора:

⁶⁴ Подробнее: <https://docs.python.org/3/install/>

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

Теперь можем запустить интерпретатор:

```
# python3.4
Python 3.4.2 (default, Oct  8 2014, 13:14:40)
[GCC 4.9.1] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import ownmod
>>> ownmod.echo()
вывод из экспортированного кода!
>>>
```

Официальная документация о расширении и встраивании интерпретатора Python:

Extending and Embedding the Python Interpreter:

<https://docs.python.org/3.6/extending/index.html>

Python/C API Reference Manual: <https://docs.python.org/3.6/c-api/index.html>

23. Python и веб-программирование на примере фреймворка Flask

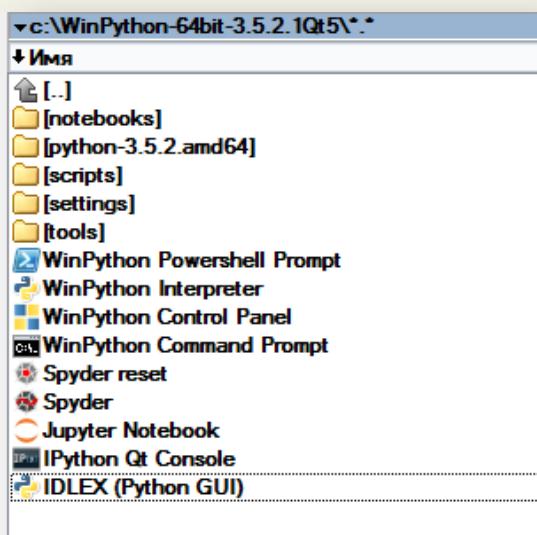
в процессе

24. Jupyter (IPython). Расширенные возможности Python

Jupyter⁶⁵ является развитием проекта IPython, который в интерактивном режиме по средствам 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>

24.1. Установка и запуск Jupyter (IPython)

Для установки Jupyter (IPython) под ОС Windows понадобится скачать и распаковать дистрибутив WinPython 3.5⁶⁶ (с сайта <https://winpython.github.io/>). После установки папка с файлами WinPython будет иметь следующий вид:

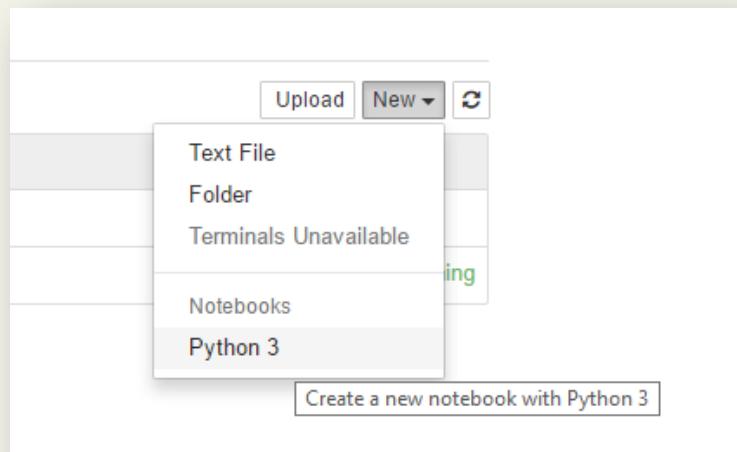


Запустим Jupyter Notebook. В процессе запуска создается локальный веб-сервер, прослушивающий сетевой порт с номером 8888. Автоматически на странице <http://localhost:8888/tree> откроется браузер.

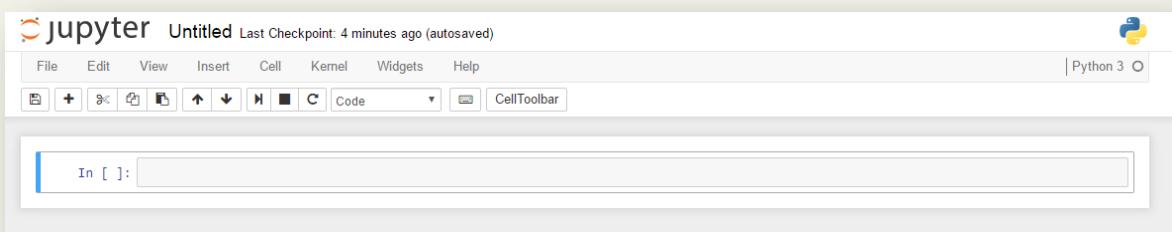
Создадим новый блокнот для запуска программ на языке Python:

⁶⁵ Официальный сайт: <http://jupyter.org>

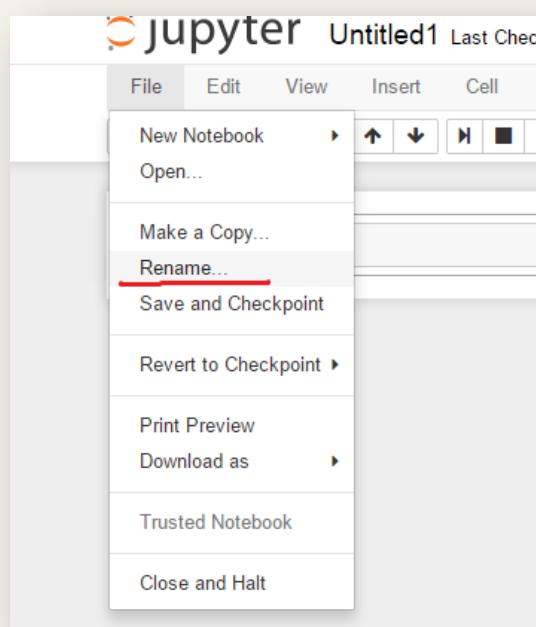
⁶⁶ Перечень пакетов, входящих в дистрибутив WinPythonQt5-3.5.2.1 представлен по ссылке: <https://github.com/winpython/winpython/blob/master/changelogs/WinPythonQt5-3.5.2.1.md>



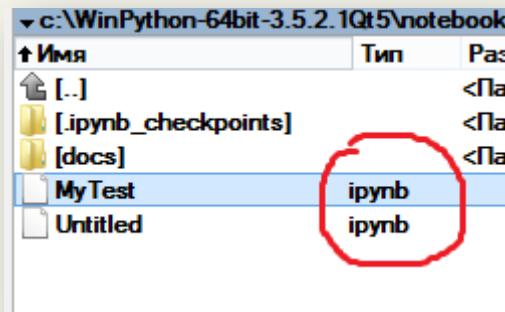
Откроется веб-интерфейс:



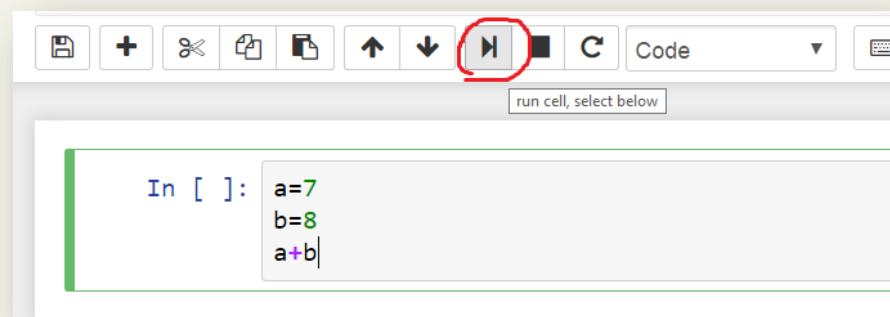
Переименуем блокнот (File → Rename) в MyTest:



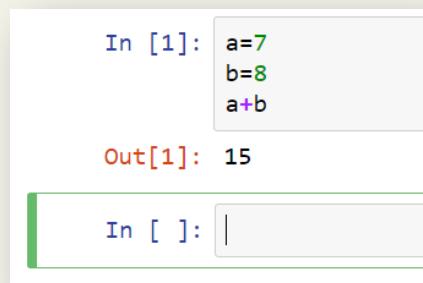
Увидим, что в каталоге \notebooks\ создался файл MyTest.ipynb:



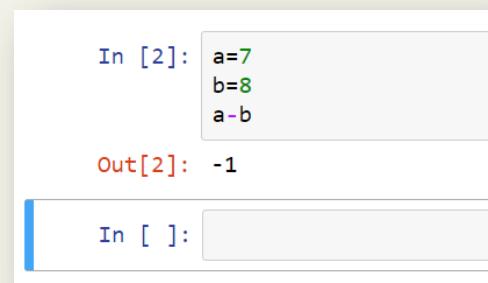
По аналогии с IDLE в ячейке In [] блокнота Jupyter набираем код на языке Python и запускаем (комбинации <Ctrl> + <Enter>, <Alt> + <Enter> – выполнить ячейку и добавить новую ячейку, <Shift> + <Enter> – выполнить ячейку и выделить следующую):



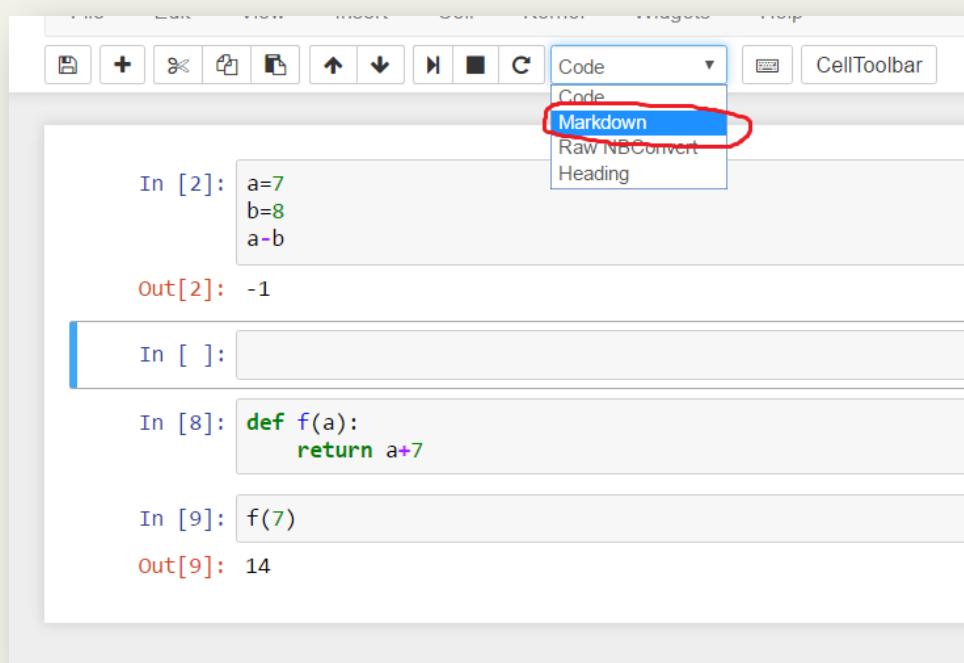
Результат выполнения кода отобразится в ячейке Out [1]:



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



Отдельные ячейки блокнота Jupyter (IPython) можно отмечать как текстовые (Markdown⁶⁷ – специальный язык разметки) для комментирования кода:

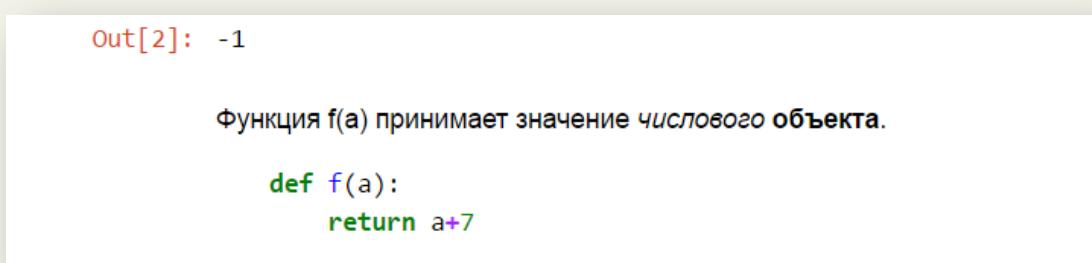


Приведем пример разметки⁶⁸:

Функция `f(a)` принимает значение *числового* **объекта**.

```
```python
def f(a):
 return a+7
```
```

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



⁶⁷ Подробнее о формате: <https://ru.wikipedia.org/wiki/Markdown>

⁶⁸ Подробнее о блокноте IPython

24.2. Работа в Jupyter (IPython)

В отличие от стандартной среды разработки IDLE Jupyter (IPython) позволяет:

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

Выполним следующий набор команд:

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

В результате получим:

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

в процессе

24.3. Интерактивные виджеты в Jupyter (IPython) 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))
```

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

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

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

x x _____ 61

61

www.english-test.net

Виджеты работают только при запущенном блокноте Jupyter (IPython).

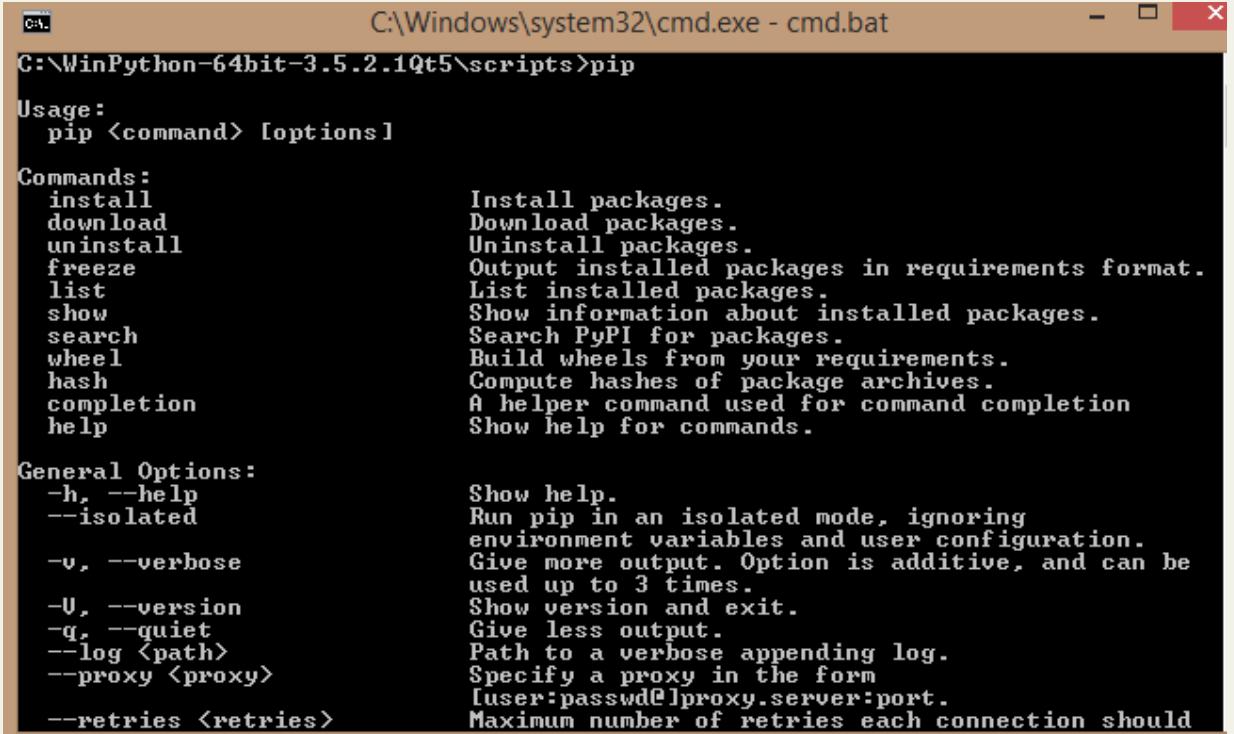
Актуальная версия документации по виджетам⁶⁹: <https://ipywidgets.readthedocs.io/en/latest/>

Примеры виджетов:

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

24.4. Установка дополнительных пакетов в WinPython из PyPI

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



```
C:\Windows\system32\cmd.exe - cmd.bat
C:\WinPython-64bit-3.5.2.1Qt5\scripts>pip
Usage:
  pip <command> [options]

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

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

⁶⁹ <https://github.com/ipython/ipywidgets/tree/master>

25. Применение Jupyter (IPython) в области защиты информации и системного администрирования

в процессе

26. Применение Jupyter (IPython) в области анализа данных (искусственного интеллекта)

в процессе

SPARQL Endpoint interface to Python (1.7.6): <https://rdflib.github.io/sparqlwrapper/>