№ 3792 МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ «МИСиС»

ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И АВТОМАТИЗИРОВАННЫХ СИСТЕМ УПРАВЛЕНИЯ

Кафедра инженерной кибернетики

А.И. Широков
М.О. Пышняк

ИНФОРМАТИКА

Разработка программ на языке программирования Питон. Базовые языковые конструкции

Учебник

Рекомендовано редакционно-издательским советом университета

МИСиС

Москва 2020

УДК 004.4
Ш64

Рецензенты:
д-р техн. наук, проф. *В.А. Поляков* РГУ нефти и газа (НИУ) им. И.М. Губкина;
канд. техн. наук, доц. *С.В. Солодов*

Широков А.И.

Ш64 Информатика : разработка программ на языке программирования Питон : базовые языковые конструкции / А.И. Широков, М.О. Пышняк. – М. : Изд. Дом НИТУ «МИСиС», 2020. – 142 с.

ISBN 978-5-907226-76-0

В учебнике рассматриваются основные конструкции языка программирования Питон (Python). Приводятся многочисленные примеры, которые позволят современному специалисту овладеть навыками для создания приложений разной направленности.

Материал соответствует учебному плану подготовки всех специальностей института ИТАСУ по разным дисциплинам бакалавриата и магистратуры, в которых применяются методы и средства алгоритмизации: «Нейронные сети», «Машинное обучение» и многие другие.

Учебник может быть использован при изучении дисциплины «Информатика», раздел «Алгоритмизация и программирование» студентами ИТАСУ, а также при освоении курса «Информатика» студентами других институтов НИТУ «МИСиС» (выбирать язык программирования).

УДК 004.4

ISBN 978-5-907226-76-0

Оглавление

ПРЕДИСЛОВИЕ

Навыки разработки программ, алгоритмическое мышление являются неотъемлемыми компетенциями современного специалиста в области информационных технологий. В предлагаемом учебнике рассмотрены все основные синтаксические элементы алгоритмического языка *Python*: типы данных (как простые, так и структурированные), операторы, правила написания функций.

До подготовки материала этой книги авторы познакомились с рядом книг [1–5]. Также были использованы источники из Интернета, ссылки на которые даны в тексте.

Алгоритмический язык *Python* (далее будем использовать еще и термин Питон) сегодня является одним из самых распространенных языков программирования. Многие авторы отмечают, что он хорошо подходит для освоения программирования и алгоритмизации.

После изучения материала учебника учащиеся будут знать основные типы данных, способы их использования, а также владеть стандартными средствами их обработки. Основное внимание в учебнике уделяется конструкциям языка программирования *Python* и умению применять их для реализации алгоритмов обработки как простых, так и комплексных данных.

Учебник содержит девять глав, снабжен многочисленными примерами, которые демонстрируют разнообразие применения языковых конструкций языка *Python*.

1. ЯЗЫК ПРОГРАММИРОВАНИЯ ПИТОН (ИСТОРИЯ, ПОПУЛЯРНОСТЬ, ФИЛОСОФИЯ, НАЗВАНИЕ)

История развития языков программирования перекликается с парадигмами, среди которых первой часто выделяют императивную. Она была реализована, прежде всего, в программах, написанных на языках машинных инструкций или Ассемблере, а в дальнейшем на *Fortran* (Фортране). Сегодня, пожалуй, бóльшая часть используемых языков программирования соответствует объектно-ориентированной парадигме. Язык программирования *Python* анонсируется как язык, поддерживающий несколько парадигм (императивную, функциональную, структурную, объектно-ориентированную). Такими свойствами обладают несколько языков.

В таблице 1.1 (основанной на Википедии и других источниках) приведены данные о нескольких языках. Некоторые из них являются сегодня такими же популярными, как и *Python*.

Таблица 1.1

Данные о языках программирования

Язык программирования	Авторы и даты разработки или появления первой версии	Основная парадигма программирования
Fortran	1954–57, Джон Бекус	Процедурный язык программирования, модульный язык программирования, язык с элементами объектно-ориентированного программирования
Algol	1958–60, комитет по языку высокого уровня при IFIP – Международной федерации по обработке информации	Процедурный язык программирования, императивный язык программирования и структурный язык программирования
Basic	1965, Томас Куртц, Джон Кемени	Алгоритмический язык программирования, позже процедурный, позже объектно-ориентированный
C	1969–1973, Денис Ритчи	Процедурный язык программирования

Окончание табл.1.1

Язык программирования	Авторы и даты разработки или появления первой версии	Основная парадигма программирования
Python	Разработка начата в конце 1980-х гг. Гвидо ван Россумом. В феврале 1991 г. опубликован исходный текст в группе новостей *alt.source* (версия 0.9.0). Версия 1.0 выпущена в 1994 г.	Мультипарадигмальный язык программирования: объектно-ориентированный язык программирования, рефлективный язык программирования, императивный язык программирования, функциональный язык программирования, аспектно-ориентированный язык программирования, динамический язык программирования
C++	1985, Бьерн Страуструп	Объектно-ориентированный язык программирования, мультипарадигмальный язык программирования, процедурный язык программирования, функциональный язык программирования, язык обобщенного программирования, язык программирования, *free-form language* и компилируемый язык программирования
Java	1995 Джеймс Гослинг, *Sun Microsystems* (в последующем приобретен *Oracle*)	Мультипарадигмальный язык программирования, язык *JVM*, отражение и язык программирования
C#	1998–2001, компания *Microsoft* под руководством Андерса Хейлсберга и Скотта Вильтаумота	Мультипарадигмальный язык программирования: объектно-ориентированный язык программирования, обобщенный язык программирования, процедурный язык программирования, функциональный язык программирования, событийный язык программирования, рефлективный язык программирования

Язык *Python* разрабатывался с конца 1980-х гг. [6, 7, 8]. Первая версия стала доступна позже многих языков программирования, представленных в табл. 1.1. Он, естественно, впитывал идеи своих предшественников. После первого сообщения о начале работы над языком до появления версии, с которой можно было бы работать, прошло несколько лет. Как и другие языки, он постоянно совершенствуется. Подтверждением этому служит тот факт, что на страницах Википедии для современных версий языков программирования, приведенных в табл. 1.1, класс не определяется одним термином (кроме *C*). Для *Python*, как и для других языков программирования, справедливо утверждение: «Новые версии языка приобретают новые возможности и свойства, соответствующие другим парадигмам программирования».

Несколько слов о названии языка *Python*. Вот что об этом говорится на странице Википедии [7]: «*Название языка произошло вовсе не от вида пресмыкающихся. Автор назвал язык в честь популярного британского комедийного телешоу 1970-х гг. «Летающий цирк Монти Пайтона». Впрочем, все равно название языка чаще связывают именно со змеей…*». Добавим, что Эрик Айдл – это «британский актер, сценарист, один из комиков группы «Монти Пайтон» [9].

А теперь приведем основной логотип языка *Python* (рис. 1.1). Это стилизованное изображение двух змей, хотя, как уже отмечалось ранее, автор языка утверждает, что название *Python* произошло от телевизионного шоу «Летающий цирк Монти Пайтона» (англ. *Monty Python's Flying Circus*). Этот знак получен с официального сайта [10].

Рис. 1.1. Логотип Питон

Кратко опишем свойства языка. Язык программирования высокого уровня *Python* является интерпретируемым в противоположность языкам компилируемым, которые превосходят первые в быстродействии. Видимо, поэтому есть несколько технологий реализации программ на нем. По одной из технологий исходный код преобразуется в специальный байт-код, который является результатом синтаксического и семантического анализа. Байт-код – это промежуточный вид программы, получаемый из исходного кода, но еще не предназначенный для автономного запуска (вне среды разработки). Имеются реализации с предварительной компиляцией в байт-код *MSIL* (*Microsoft*) или *Java*. Также есть средства, формирующие на основе исходного кода *Python* исполнимый код, выполняемый вне среды разработки.

Часто среди свойств рассматриваемого языка называют его ориентированность на повышение производительности разработки (не выполнения) и читаемости кода. Несомненным достоинством этого языка является его многоплатформенность. На странице Википедии приводится такое описание этого факта [6]: «*Python портирован и работает почти на всех известных платформах — от КПК до мейнфреймов. Существуют порты под Microsoft Windows, практически все варианты UNIX (включая FreeBSD и Linux), Plan 9, Mac OS и Mac OS X, iPhone OS 2.0 и выше, Palm OS, OS/2, Amiga, HaikuOS, AS/400 и даже OS/390, Windows Mobile, Symbian и Android*».

Имеется более десятка версий языка [7]. Версия с номером *Python* 1.0 выпущена в январе 1994 г., а последняя – *Python* 3.7 – в июне 2018 г. В данном учебнике примеры опробованы на версии *Python* 3.8.0.1a 2019 г. (*pre-release*). Обзор возможностей последней версии можно найти на сайте [11].

Нет никаких особенных ограничений на условия распространения программ на этом языке, точнее они должны соответствовать специальной лицензии [12]: «*Python Software Foundation License* (*PSFL*) – *BSD-подобная пермиссивная лицензия на свободное ПО, совместимая с GNU General Public License* (*GPL*). *Ее первичное назначение – распространение программного проекта Python. В отличие от GPL, лицензия Python не является копилефтной и позволяет вносить изменения в исходный код, а также создавать производные работы, не открывая код.*

PSFL указана как утвержденная как FSF, так и OSI». В соответствии с этой лицензией нет ограничений на коммерческое использование программ, написанных на этом языке. Более подробная информация будет приведена далее.

Создатели языка программирования *Python* следуют определенной философии, созданной Тимом Петерсом (*Tim Peters*). Она базируется на *The Zen of Python* («Дзен Пито́на», или «Дзен Па́йтона»). По команде ***import this*** получим основные положения философии (рис. 1.2).

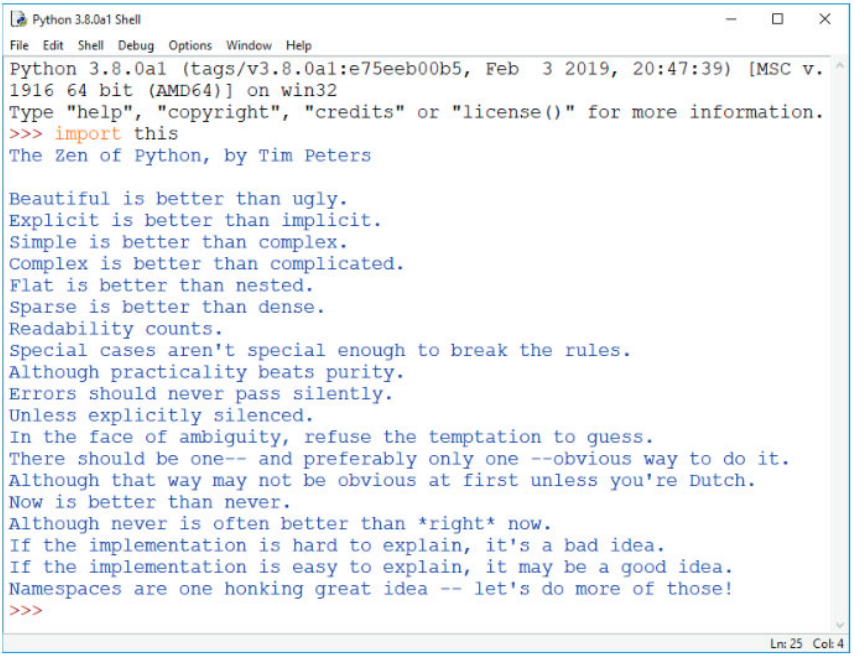


Рис. 1.2. Философия *Python*

Приведем ее перевод [6].

Красивое лучше, чем уродливое.
Явное лучше, чем неявное.
Простое лучше, чем сложное.
Сложное лучше, чем запутанное.
Плоское лучше, чем вложенное.

Разреженное лучше, чем плотное.

Читаемость имеет значение.

Особые случаи не настолько особые, чтобы нарушать правила.

При этом практичность важнее безупречности.

Ошибки никогда не должны замалчиваться.

Если не замалчиваются явно.

Встретив двусмысленность, отбрось искушение угадать.

Должен существовать один – и желательно только один – очевидный способ сделать это.

Хотя он поначалу может быть и не очевиден, если вы не голландец.

Сейчас лучше, чем никогда.

Хотя никогда зачастую лучше, чем прямо сейчас.

Если реализацию сложно объяснить – идея плоха.

Если реализацию легко объяснить – идея, возможно, хороша.

Пространства имен – отличная вещь! Давайте будем делать их больше!

Эти полушутливые фразы призывают программистов при написании исходного текста придерживаться определенного стиля. Эти положения, выраженные в абстрактной форме, дополняет свод правил написания программ *PEP*8 [13].

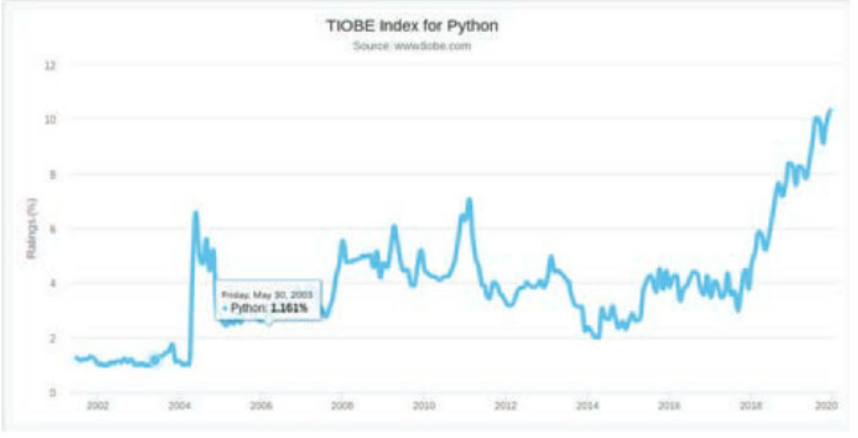


Рис. 1.3. Динамика индекса популярности *Python*

В последние годы в списке популярности языков программирования *Python* регулярно занимает высокие позиции. Приведем данные, опубликованные на сайте *tiobe* [8], на который часто ссылаются многие авторы, анализируя положение дел в этом вопросе. В декабре 2019 г. *Python* занимает третье место с рейтингом 10% после *Java* (17%) и *C* (16%). На рисунке 1.3 приведен рейтинг этого языка по годам [14], что показывает положительную динамику его популярности за последние 14 лет. Анализируя график, кроме общей тенденции неуклонного возрастания популярности, можно отметить некоторое ослабление интереса к языку в двух периодах (2006–2008 и 2014–2016).

Отметим, что рейтинг упомянутого ранее *Java* за период с 2002 по 2019 г. постоянно уменьшается (рис. 1.4). Но его рейтинг все еще самый высокий среди всех языков.

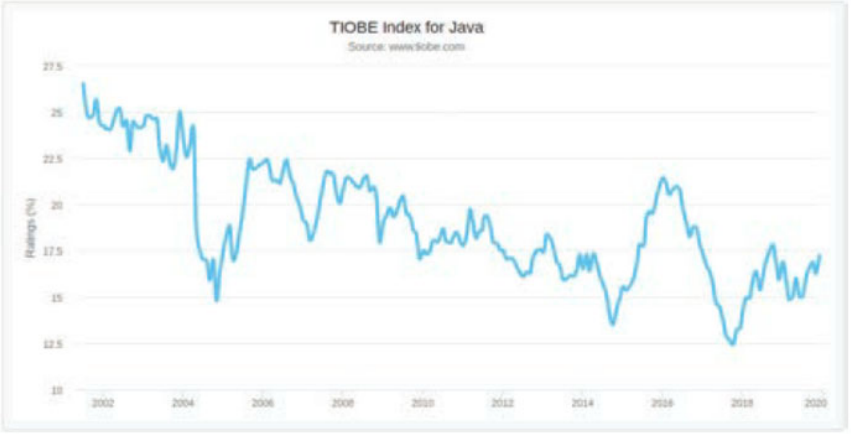


Рис. 1.4. Динамика индекса популярности *Java*

Выше говорилось, что в июле 2019 г. язык программирования *C* по популярности расположился между *Java* и *Python*. Он обгоняет даже своих более «молодых коллег» *C++* и *C#*. Но общая тенденция его популярности (как и *Java*) неуклонно снижается с 2002 г., хотя всплески интереса к нему наблюдались в 2004 и 2014 гг. Это демонстрирует рис. 1.5.

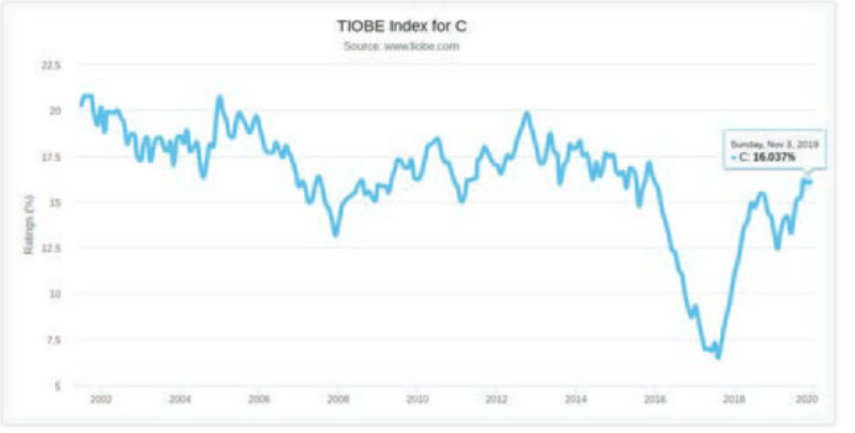


Рис. 1.5. Динамика индекса популярности *C*

Python, в отличие от некоторых своих конкурентов не имеет явного приемника, как, например, *C++* или *Java*. Он создан энтузиастами, один из которых работал в проекте по созданию языка программирования *ABC*.

Перейдем к описанию средств, предоставляемых программистам, которые работают с Питоном.

Экзаменационные темы

1. Название языка программирования *Python*.
2. Характеристика языка программирования *Python*.
3. Популярность языка *Python*.

2. СРЕДА IDLE И ПЕРВЫЕ ДЕЙСТВИЯ В РЕЖИМЕ ИНТЕРПРЕТАТОРА

2.1. Среда IDLE: название, библиотека Tkinter, начальный экран

Есть две возможности использовать язык программирования *Python*: выполнять отдельные команды в оболочке («покомандный» режим) или набрать весь исходный текст программы, а затем выполнить его целиком.

Вот что написано о среде разработки *Python*, обеспечивающей эти возможности [15]: «***IDLE*** (***I****ntegrated* ***D****evelopment and* ***L****earning* ***E****nvironment* [1]) – *это интегрированная среда разработки и обучения на языке Python, созданная с помощью библиотеки Tkinter. Официально – искажение IDE, но на самом деле названа в честь Эрика Айдла (англ. Eric Idle) из Монти Пайтон. Поставляется вместе с Python и благодаря библиотеке Tkinter может использоваться на многих платформах, среди которых Windows, Mac OS, Unix-подобные ОС* [7]».

Однако есть и другая трактовка того, как расшифровывается *IDLE*. В источнике [16] она такая: ***I****ntegrated* ***D****eve****L****opment* ***E****nvironment*. Заметим, что первая трактовка названия содержится и на официальном сайте [11], и в позиции меню *Help-About IDLE* (рис. 2.1).

На рисунке 2.1 содержится упоминание о *Tk*. Эта библиотека, имеющая полное название *Tkinter*, позволяет создавать приложения с графическим интерфейсом. На странице Википедии о ней говорится следующее [17]: «*Tkinter* (*от англ. Tk interface*) – *кросс-платформенная графическая библиотека на основе средств Tk* (*широко распространенная в мире GNU/Linux и других UNIX подобных систем, портирована также и на Microsoft Windows*), *написанная Стином Лумхольтом* (*Steen Lumholt*) *и Гвидо ван Россумом* [1]. *Входит в стандартную библиотеку Python*». Она поставляется вместе с *Python* и может быть использована для написания дружелюбных к пользователю программ. В приложении 1 приводится исходный текст программы, выводящий в окно сообщение «*Hello, world*».

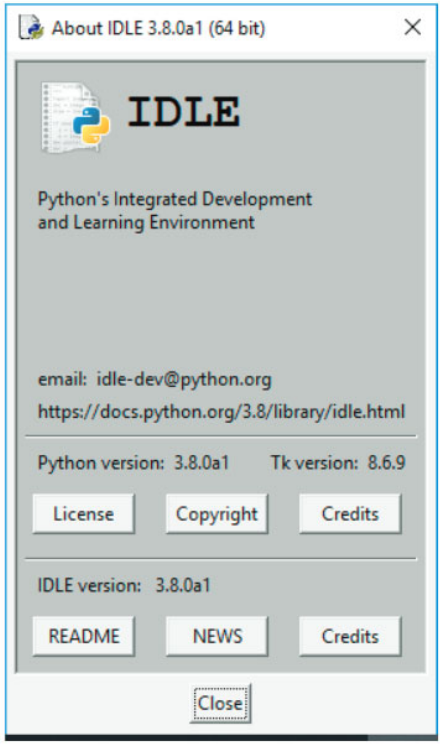


Рис. 2.1. Диалоговое окно *About IDLE*

При вызове среды *Python* на компьютере с операционной системой *Windows* появляется такое окно (рис. 2.2).

В первых строках размещена информация о версии *Python* (3.8.0a1), далее – о компьютерной платформе. Заключительная строка приглашает набрать такие команды: *help*, *copyright*, *credits* и *license*(). Обзор команд меню оболочки будет приведен позже. Здесь приведем перечень возможностей, которые представлены после выбора закладки *General* в диалоговом окне *Setting*, появляющемся после выбора команд *Options-Configure IDLE*. Наверное, главная возможность этого окна – выбрать режим работы со средствами языка Питон: исполнять одиночные команды или набрать весь текст программы, а затем выполнить его целиком. Это демонстрируется рис. 2.3. Для фиксации пер-

вого надо выбрать *Open Shell Windows* (покомандный режим), а второго – *Open Edit Windows* (режим среды разработки). Там же есть возможность задать целый ряд свойств вызываемого окна.

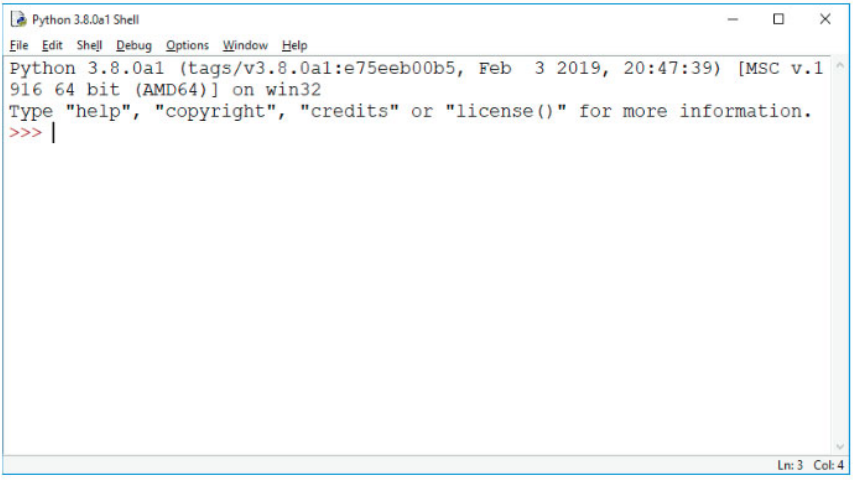


Рис. 2.2. Начальное окно *IDLE*

Вернемся к вопросу переносимости *Python* на разные компьютерные платформы. Здесь приведем информацию о том, что выводится при вызове *Python* на компьютерах с операционной системой *MacOS* и *Linux*. Есть разные технологии разработки программ на *Python*. На приведенном скриншоте (рис. 2.4) представлен начальный экран интегрированной среды разработки для операционной системы *MacOS*. Программное средство такого типа содержит как минимум текстовый редактор. Он обеспечивает формирование исходного текста программы. Далее этот текст транслируется в машинный код, а затем к последнему подключаются необходимые библиотечные функции. Результатом таких действий будет исполнимый модуль (готовая для запуска программа), который выполнятся без запуска среды разработки. Но такие программы могут включать в себя систему выполнения успешно оттранслированной программы и средства для удобной отладки алгоритмов. Некоторые *IDEL* позволяют разрабатывать программы на нескольких языках программирования и даже

формировать исполнимые модули для разных компьютерных платформ.

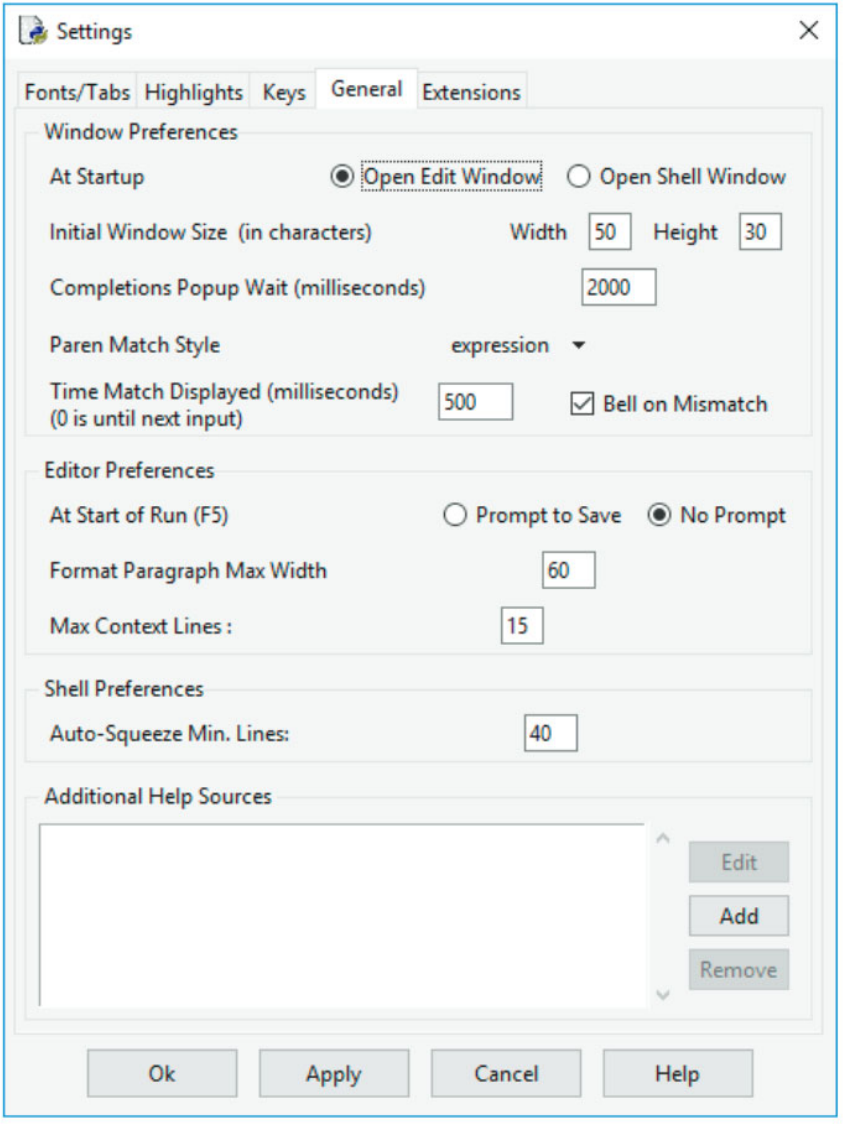


Рис. 2.3. Закладка *General* в диалоговом окне *Setting*

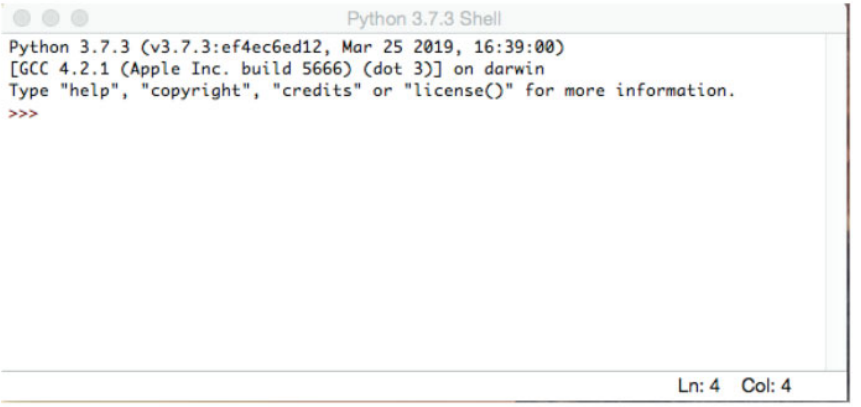


Рис. 2.4. Скриншот *IDLE Python* на *MacOS*

Следующий скриншот (рис. 2.5) содержит экран, на котором показано, что выводится, когда запущена оболочка *Python* на компьютере с операционной системой *Linux*. В таком режиме можно выполнять отдельные команды этого языка. Если нужно отработать множество строк с конструкциями этого языка, надо сформировать исходный текст *Python* в другой программе – текстовом редакторе (наподобие блокнота *Windows*).

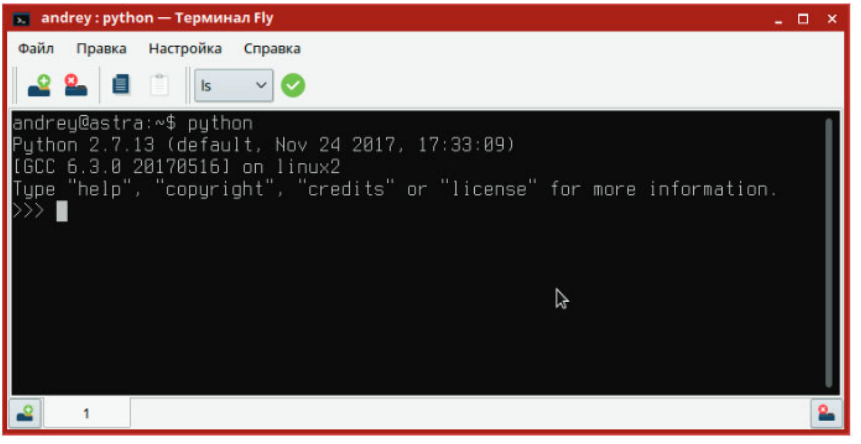


Рис. 2.5. Скриншот *Python* на *Linux*

Сообщим, что приведенный скриншот получен после запуска восьмой версии операционной системы *Alt Linux*. Образование – в режиме *Live CD* (без установки на компьютер).

Есть еще одна возможность создавать программы на Питоне и изучать возможности этого языка – онлайн-компиляция. Таких ресурсов Интернете много. Приведем адрес одного из них – https://rextester.com/l/python3. На начальной странице этого ресурса в редакторе кода приведена программа *Hello*.

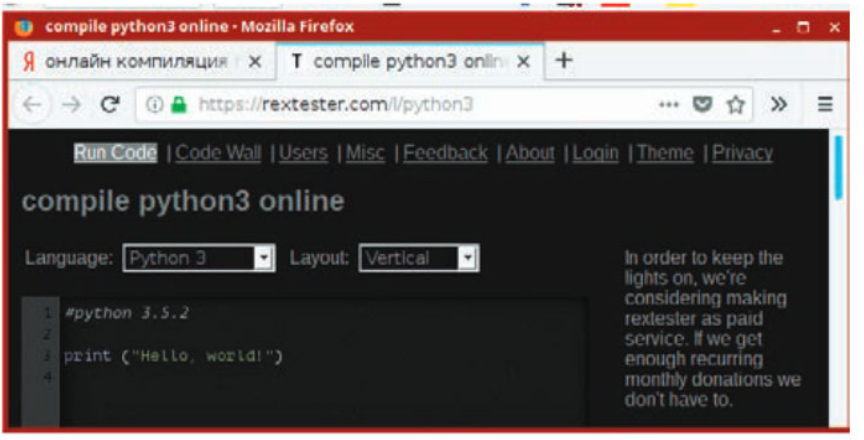


Рис. 2.6. Онлайн компиляция программы на *Python*

Активируя кнопку *Run it* (F8) в окне, расположенном ниже, увидим результат.

Теперь перейдем к первым действиям, которые можно совершать в оболочке *Python*.

2.2. Описание команд help, copyright, credits и license()

Опишем, что выводится при выполнении команд начального экрана *IDLE*, упомянутых ранее. Первой опишем, какую информацию получим после выполнения команды *license*(). Это текст лицензии *PSFL* (*Python_Software_Foundation_License*), но не только. Вывод начинается с краткой истории *Python* и опи-

сания того, какие версии *Python* совместимы с лицензией *GPL*. Информацию о последней и ее текст на английском языке можно получить по ссылке [18]. *GPL* создана Ричардом Столменом – основателем движения *free software* (свободное программное обеспечение). В рамках этого движения действует фонд – *FSF* (*Free Software Foundation*). Аналогично для *Python* создан и действует *PSF* (*Python_Software_Foundation*). Как было сказано ранее, при выполнении команды *license*() выводится текст лицензии, поддерживаемый этим фондом.

Другая команда (*help*) выводит справочную информацию об объектах *Python*. Если ее исполнить без параметров, выводится информация, представленная на рис. 2.7.

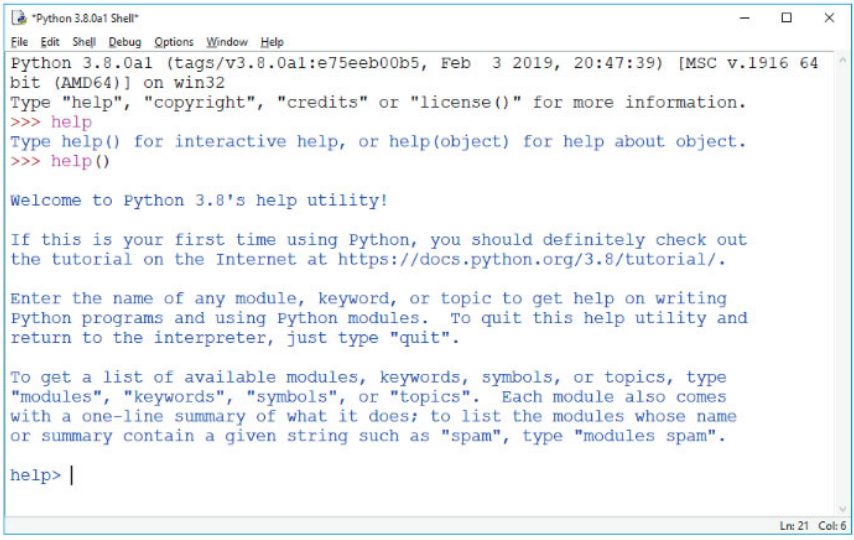


Рис. 2.7. Команда *help*

Здесь указано, как работать с утилитой, но сначала дается ссылка на обучающую страницу. Для получения помощи надо в скобках задать имя интересующего объекта (модуля, ключевого слова или раздела документации). Например, *help*(*input*) выведет следующее (рис. 2.8).

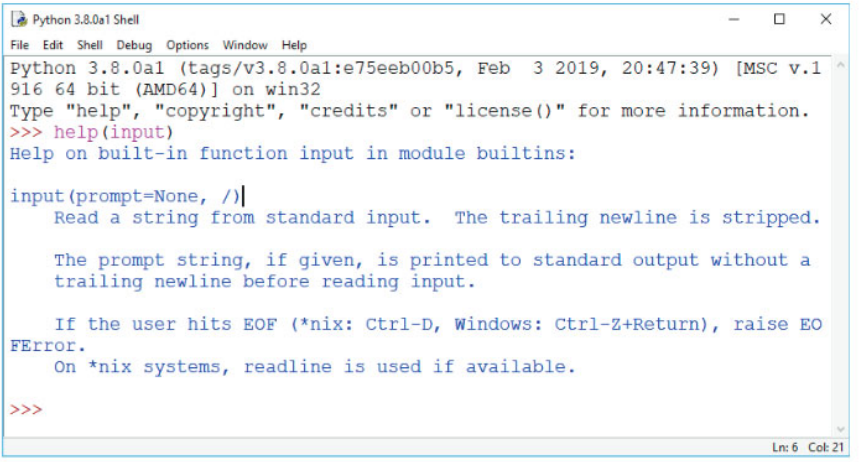


Рис. 2.8. Помощь по команде *input*

Для получения помощи об операторах *Python* их имена надо вводить в кавычках. После ввода *help*('*for*') сначала выводится сообщение о 59 строках запрашиваемого текста (рис. 2.9). Если кликнуть в области сообщения два раза мышкой, выведется информация об операторе.

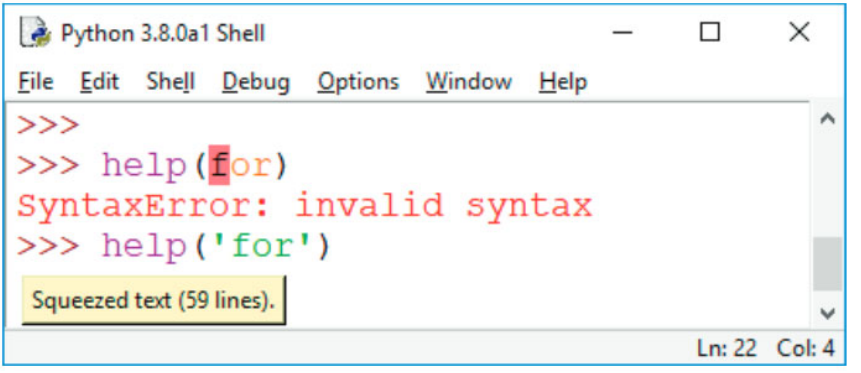


Рис. 2.9. Команда *help*('*for*')

Команда *copyright* фиксирует права на язык *Python*. Из рисунка 2.10 видно, что владелицей марки *Python* были четыре организации, а с 2001 г. по настоящее время – фонд *PSF*.

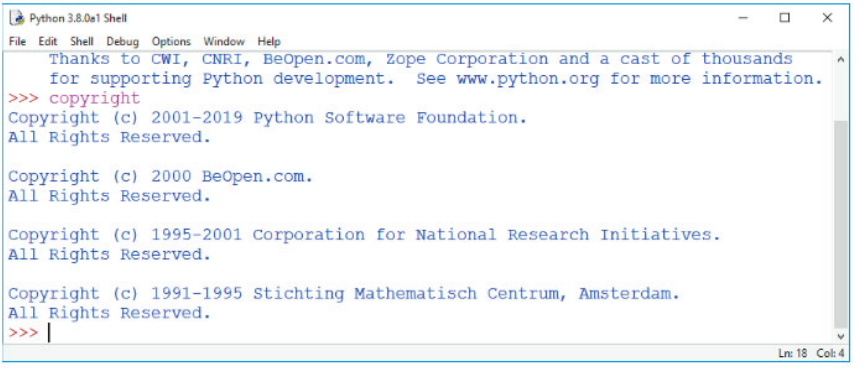


Рис. 2.10. Команда *copyright*

И в заключении приведем информацию (рис. 2.11), выводимую командой *credits*.

Рис. 2.11. Команда *credits*

Выдается информация об организациях, в которых разрабатывались средства языка Питон. Сообщается, что в их поддержке принимают участие тысячи авторов. Приводится адрес основного сайта.

2.3. Меню IDLE

Теперь кратко опишем основное меню оболочки *Python*. Оно имеет традиционную структуру: начинается с *File*, а заканчивается *Help*. В этом можно убедиться, посмотрев на любой из рис. 2.6–2.11. Всего в главном горизонтальном меню семь позиций. Не будем подробно описывать все. В таблице 2.1 приведены три команды из выпадающих меню.

Таблица 2.1

Несколько команд оболочки *Python*

Команда и комбинация горячих клавиш	Назначение
Позиция меню *File*	
New File ... Ctrl+N	Открыть новое окно
Open ... Ctrl+O	Открыть существующий файл Питон
Recent Files	Открыть сохраненный ранее файл
Save ... Ctrl+S	Сохранить файл
Save As ... Ctrl+Shift+S	Сохранить как
Close ... Alt+F4	Закрыть окно редактора
Exit ... Ctrl+Q	Выход
Позиция меню *Edit ( )*	
Undo	Назад
Redo	Вперед
Cut	Вырезать
Copy	Копировать
Paste	Вставить
Select All	Выбрать все
Find	Найти
Find Next	Найти следующее
Previous History ... Alt+P	Вызов в строку оболочки последней исполненной команды (можно вернуть несколько)

Переходим к описанию первых действий в Питоне. Сначала изучим их в режиме оболочки.

2.4. Первые действия в режиме оболочки

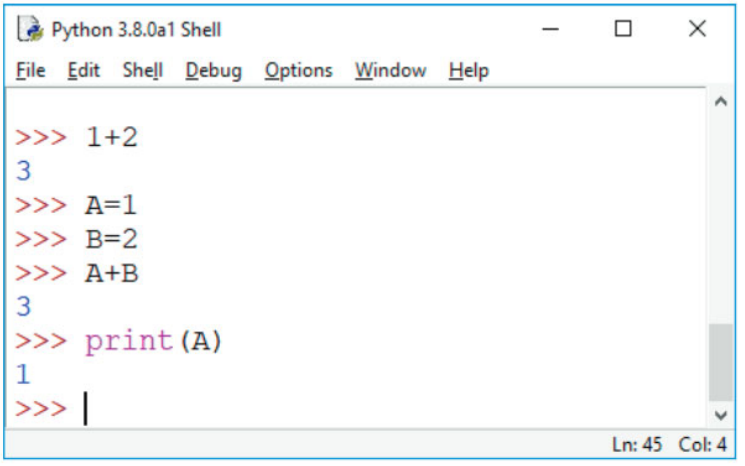
В начале этого раздела сделаем оговорку о том, что в дальнейшем будем использовать вместо слова *Python* его русский аналог – Питон. Среда Питона позволяет делать многие действия, используя как неизменяемые значения (константы), так и переменные.

Например, если ввести следующее арифметическое выражение 1+2, то результатом будет 3.

Можно сохранить значения в переменных. Своим именем они связаны с памятью, в которой значение сохраняется в течение сеанса. Такие значения в Питоне определяются операцией присваивания. Если последовательно набрать такие команды, то получим также 3.

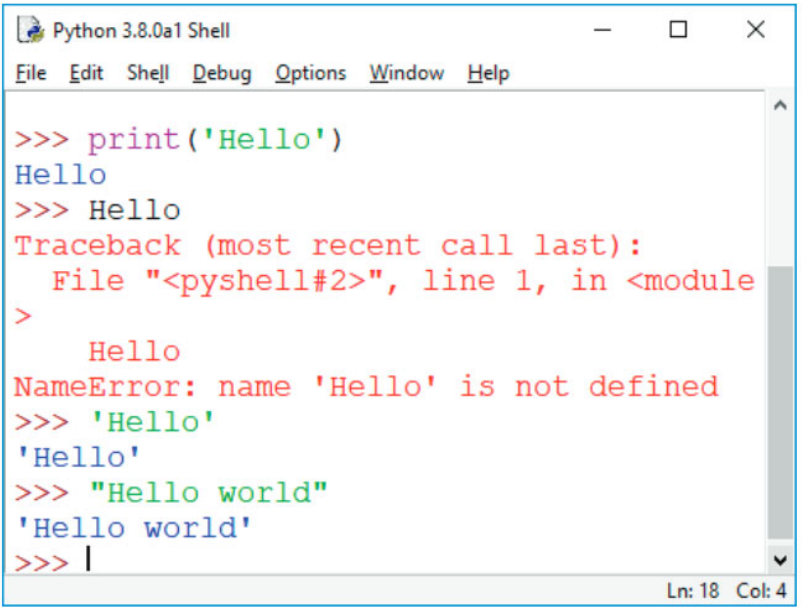
```
A=1
B=2
A+B
```

Заметим, что далее у скриншотов подрисуночные надписи приводиться не будут.



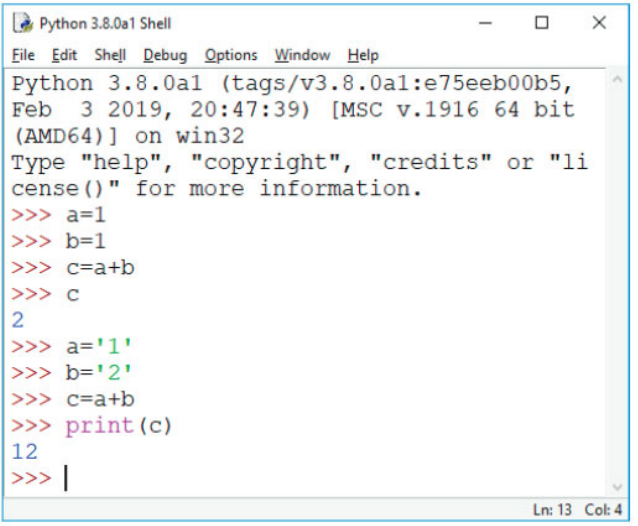
В Питоне многие действия можно выполнить разным способом. Продемонстрируем это. Для вывода значений переменной в режиме оболочки можно либо просто указать ее имя, либо выполнить *print*().

Если ввести произвольный набор символов, не являющийся выражением, значение которого можно вычислить, то выводится сообщение об ошибке. Наборы символов (слова) следует оформить как символьные (иногда используют строковые, от англ. *string*) значения (константа, литерал). Для этого используются двойные или одинарные кавычки.

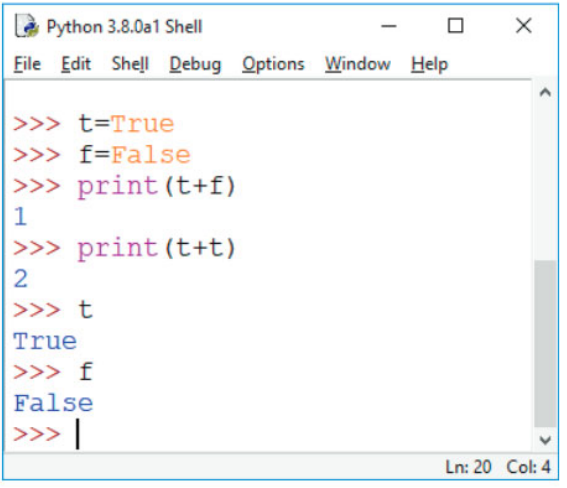


При запуске оболочки разные элементы диалога выводятся разным цветом. Например, приглашение вводить команды (три символа больше >) имеют коричневатый цвет. Вводимый текст команд будет черным, результат работы команд – синим, а сообщения об ошибках и комментарии – красного цвета. Фиолетовым цветом выделяются ключевые слова. Такое закрепление цветов может быть изменено.

Питон поддерживает несколько типов данных. Далее они будут рассмотрены подробнее. Для каждого из них определен свой набор операций. Например, операция, задаваемая знаком «+», не всегда приводит к арифметическому действию сложения.



Для логических данных операция «+» приводит к такому результату.



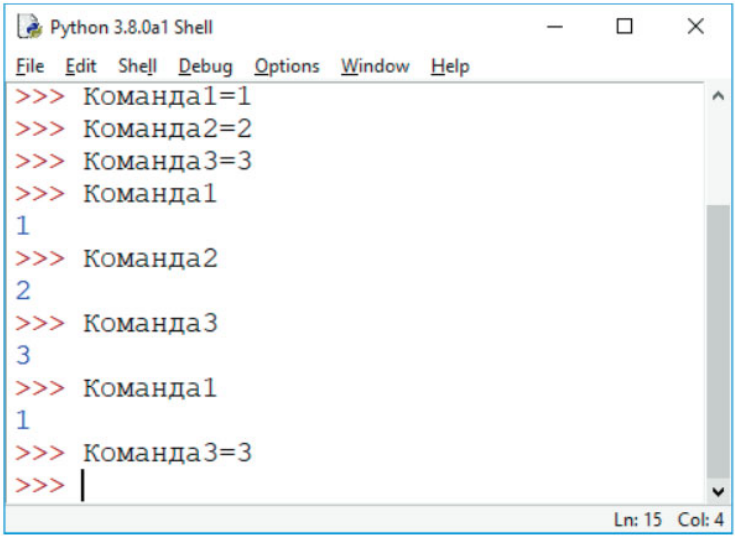
Видно, что при выполнении операции «+» логическое значение Истина (*True*) интерпретируется как 1, а логическое значение Ложь (*False*) как 0.

В оболочке можно возвратить набранные ранее команды. Для этого надо использовать комбинацию <Alt+p> или <Alt+N>.

Введем такие команды.

Команда1=1
Команда2=2
Команда3=3
Команда1
Команда2
Команда3

Далее выполним комбинацию клавиш <Alt+p> три раза и <Alt+n> три раза. Получим следующее.



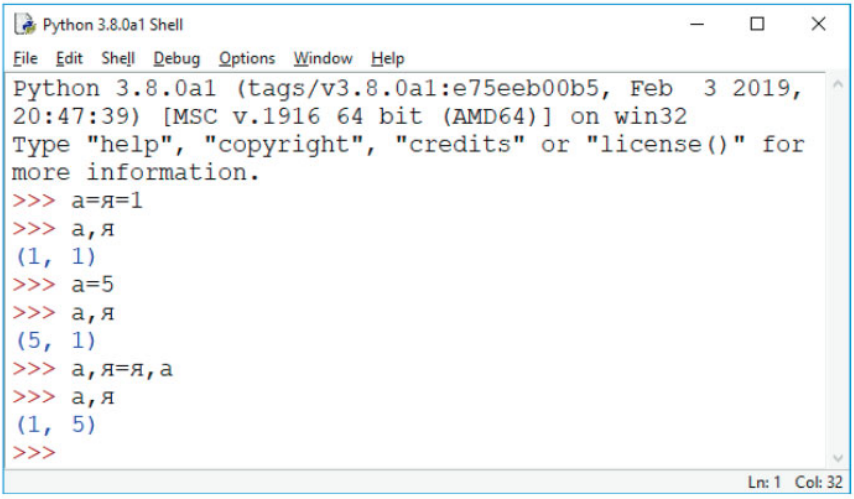
Выполнив <Alt+p> три раза в строке приглашения, получим Команда1. Ее можно выполнить. А нажатие <Alt+n> приведет к появлению Команда3=3 в строке набора команд.

Наверное, в программировании чаще других встречается операция присваивания значений переменным. В разных языках это действие может иметь свои особенности. В Питоне, как было показано выше, эта операция «порождает» переменную и фиксирует ее тип. Есть возможность множественного присваивания, что демонстрирует следующий пример.

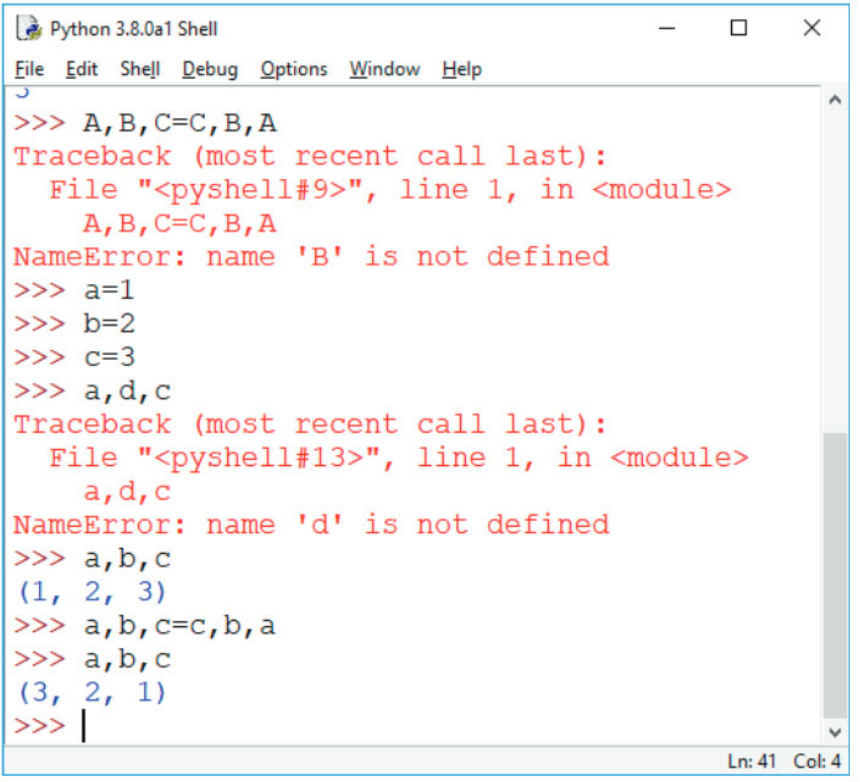
```
а=я=1
а,я
# Будет (1, 1)
я=5
а,я
# Будет (1, 5)
```

Знак операции присваивания можно использовать для обмена значений переменных.

```
# Обмен значений переменных
а,я=я,а
а,я
# Будет (5, 1)
```



Еще обмен значениями демонстрирует следующий пример. В нем меняются значения трех переменных.



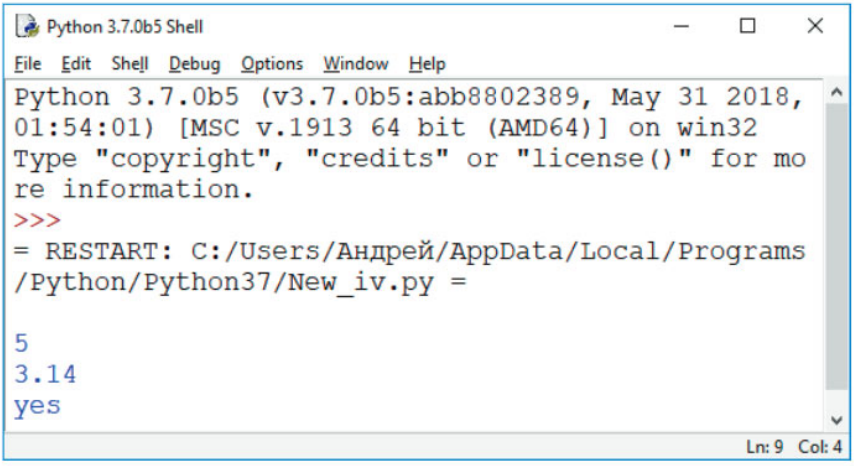
Рассмотрим важный в теории программирования момент, связанный с реализацией свойств типов данных, поддерживаемых в Питоне.

2.5. Динамическое переопределение типа

В теории языков программирования есть два противоположных понятия: статическая и динамическая типизации. При первом для переменной тип фиксируется при ее определении и далее не может изменяться. А при втором тип переменной определяется при присваивании ей значения. Таким образом, тип переменной может изменяться при выполнении программы несколько раз. Питон является языком программирования с динамической типизацией. Продемонстрируем это. Будем по-

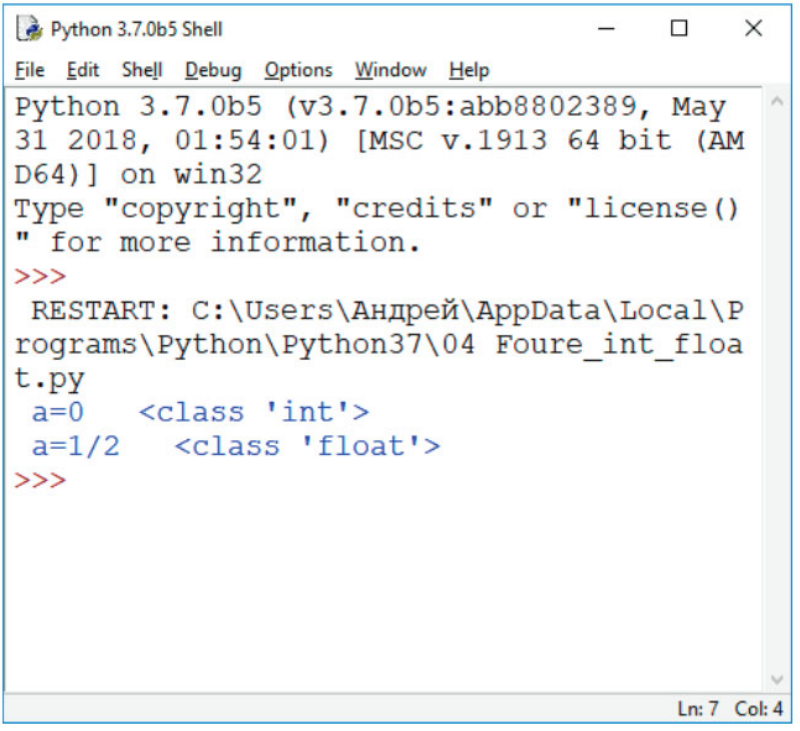
следовательно присваивать переменной целое, вещественное и символьное значения и сразу выводить их.

```
a = 5
print(a)
a=3.14
print(a)
a='yes'
print(a)
```



Узнать тип переменной можно, используя функцию *type*. Приведем пример ее использования.

```
a=0
print(" a=0 ", type(a) )
a=1/2
print(' a=1/2 ', type(a) )
```



В следующем разделе рассмотрим, как в Питоне реализуются циклы.

2.6. Циклы и инструмент range

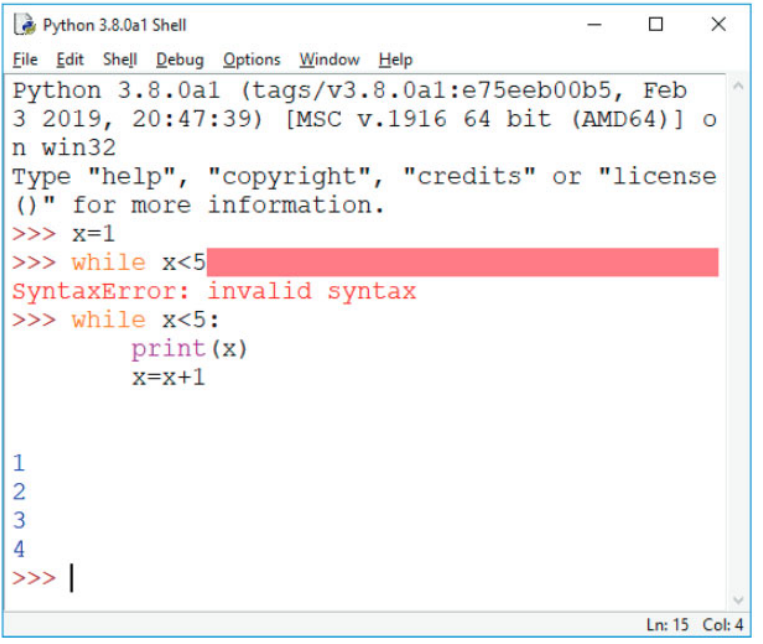
В языке Питон циклы реализуются, в отличие от других языков, только двумя конструкциями: *while* и *for*. В первом примере организован цикл, выводящий значения переменной x, которая изменяется в цикле от 1 с шагом 1 и пока ее значение меньше 5.

```
>>> x=1
>>> while x<5
SyntaxError: invalid syntax:
>>> while x<5:
        print(x)
        x=x+1
```

Действия выполняются после ввода, когда в строке ничего не набрано. Будет выведено следующее.

1
2
3
4

Здесь приведено сообщение об ошибке в записи оператора *while* (отсутствует символ).

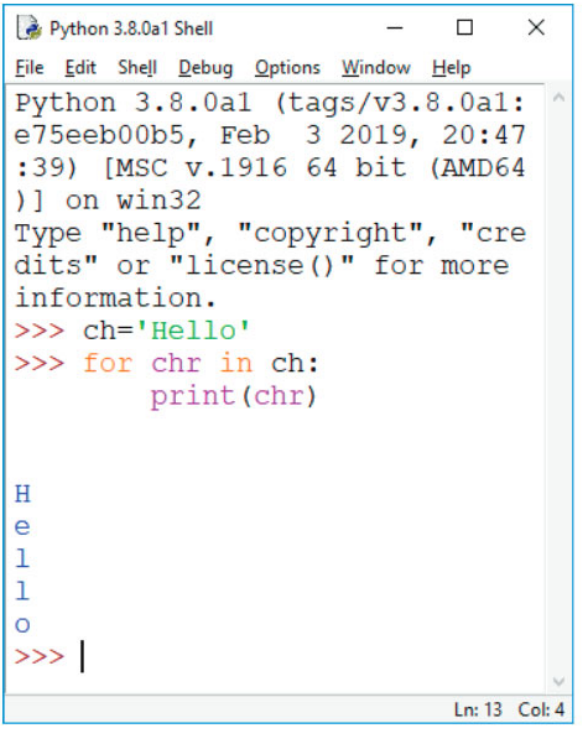


Другая циклическая конструкция использует ключевое слово *for*. В отличие от некоторых языков программирования в Питоне *for* реализует перебор всех элементов объекта (набора), записанного после слова *in*. В следующем примере таким объектом является строка *'Hello'*.

```
# Циклы, теперь for
>>> ch='Hello'
>>> for chr in ch:
    print(chr)
```

Будет выведено:

H
e
l
l
o

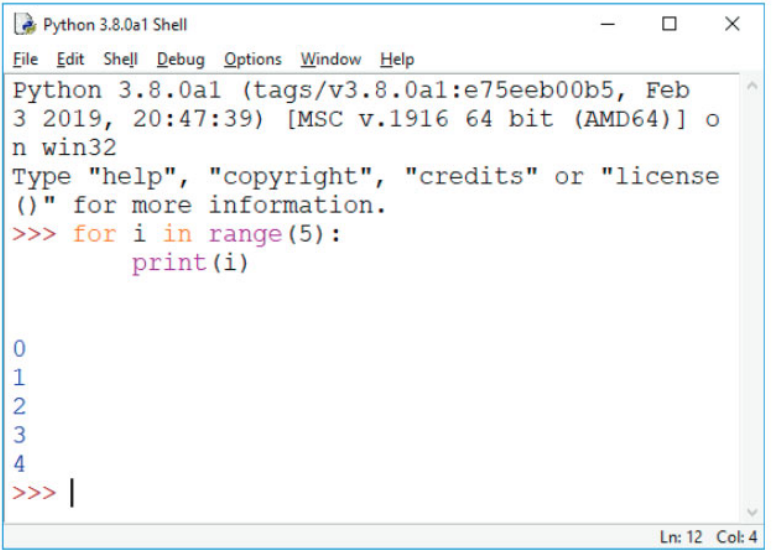


В конструкции *for* может быть использован инструмент *range*. Фактически это функция, генерирующая множество целых чисел, вычисляемая на основе начального и предельного значения, а также шага приращения значений. В общем виде такая функция выглядит следующим образом:

range(start, stop, step)

Функция *range* генерирует целые числа так, чтобы значение *stop* не достигалось. Из трех параметров рассматриваемой функции обязательным является только *stop*.

```
>>> for i in range(5):
        print(i)
0
1
2
3
4
```



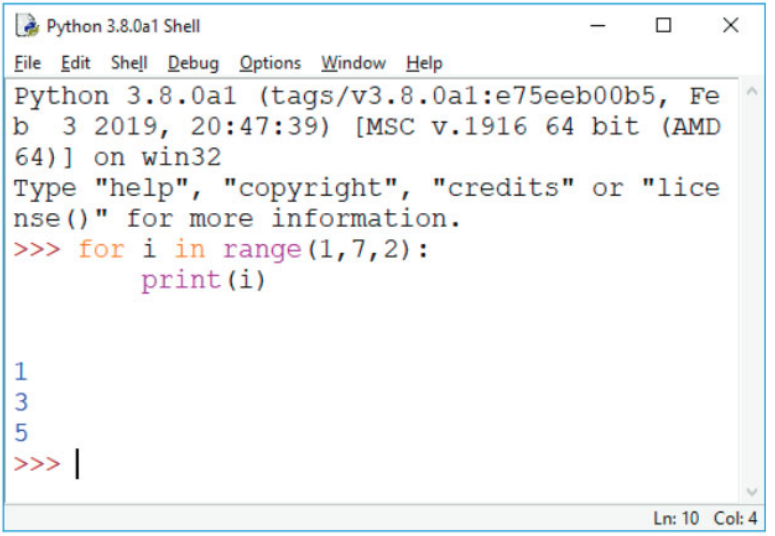
Во втором примере заданы только начальное и конечное значения.

```
# Инструмент range
>>> range(1,5)
range(1, 5)
>>> for i in range(1,5):
        print(i)
1
2
```

```
3
4
```

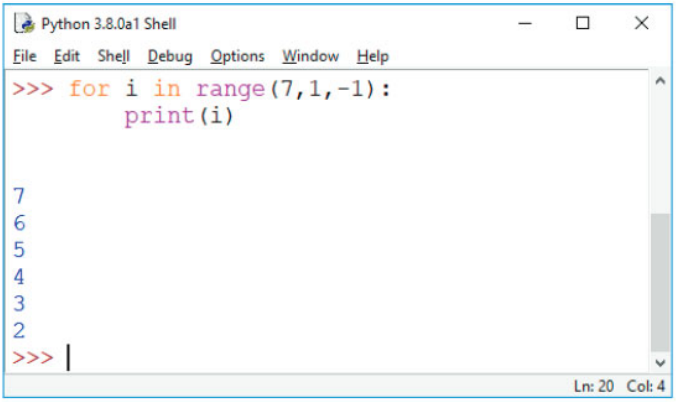
Из этих примеров ясно, что значение *step* по умолчанию равно +1. Но ему можно задавать и другие значения, например +2, как в этом примере.

```
>>> for i in range(1,7,2):
        print(i)
1
3
5
```



Теперь рассмотрим пример, в котором инструмент *range* использован с отрицательным шагом.

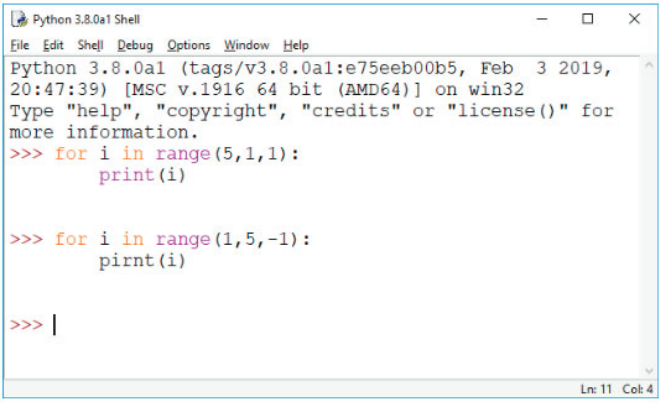
```
for i in range(7,1,-1):
    print(i)
7
6
5
4
3
2
```



Приведем еще два примера использования функции *range* с разными последним значением и шагом.

```
for i in range(5,1,1):
    print(i)
for i in range(1,5,-1):
    print(i)
>>>
```

Как видно, в обоих примерах цикл не сработает ни разу – вывод пустой.



Далее будут рассмотрены примеры реализации условного алгоритма в Питоне.

2.7. Условный оператор

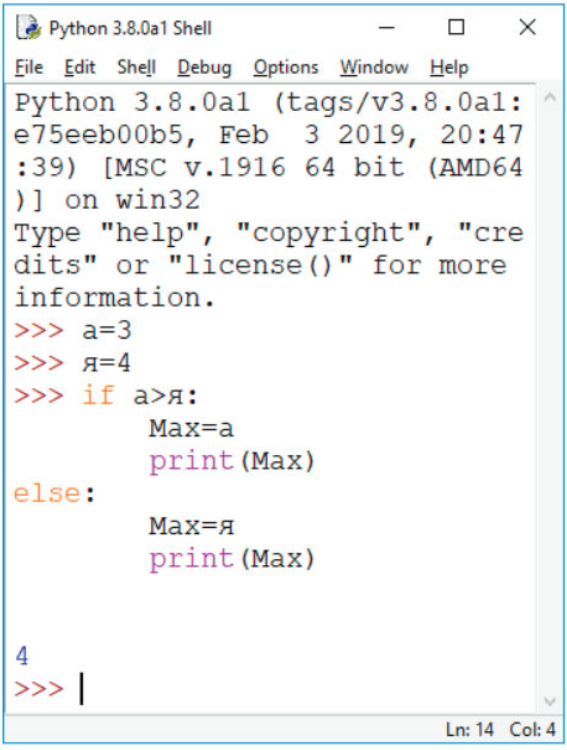
Условный оператор также может быть реализован в режиме оболочки. Рассмотрим, как можно его реализовать для решения такой задачи: из двух заданных чисел выбрать наибольшее. В первом варианте показано, что после ввода строки с ключевым словом *if* в следующей строке автоматически набор начинается после отступа (табуляции). Можно ввести действия, которые выполняются при истинности условий. Для завершения такого списка операторов надо нажать клавишу *Backspace*. Если этого не сделать, то выводится сообщение об ошибке.

```
>>> a=3
>>> b=4
>>> if (a>b):
        Max=a
else: # Надо нажать Backspace
        Max=b
print(Max)
SyntaxError: invalid syntax
>>> Max
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    Max
NameError: name 'Max' is not defined
```

Вот так сработает этот оператор.

```
>>> a
3
>>> b
4
>>> if (a>b):
        Max=a
else:
        Max=b
>>> Max
4
```

Или так.



В заключение отметим, что в этом разделе коротко рассмотрено множество языковых конструкций Питона. Их возможности демонстрировались в режиме оболочки. Далее они будут рассмотрены более подробно в среде разработки.

Экзаменационные темы

1. Среда разработки *Python* (*IDLE*).
2. Команда *help*.
3. Команда *license*.
4. Меню *IDLE*.
5. Типы данных Питона, переопределение типа.
6. Инструмент *range* и его использование в цикле *for*.

3. КАК ПРОГРАММЫ НА ПИТОНЕ ОБЩАЮТСЯ С «ВНЕШНИМ МИРОМ»

Как было сказано ранее, средства языка Питон можно использовать в двух режимах. Первый – ввод отдельных команд в оболочке. Во втором исходный текст набирается целиком в редакторе, аналогичном Блокноту *Windows*, затем исполняется целиком. Все примеры, которые далее рассматриваются в учебнике, исполнены во втором режиме.

В меню редактора текстов Питона семь команд: *File–Edit–Format–Run–Option–Windows–Help*. Пять из них совпадают с командами меню оболочки. Но третья и четвертая команды отличаются: *Format* (вместо *Shell*) и *Run* (вместо *Debug*). В меню *Run* выделим команду *Run*, которая выполняется при нажатии клавиши F5.

В этом разделе опишем, как программы, написанные на языке Питон, общаются с «внешним миром». В данном случае под общением понимается операция действия по выводу информации пользователю программы и получение от него каких-то данных (чаще всего исходных для дальнейшего расчета). Под программой в данном случае подразумевается исходный код, сохраненный в текстовом файле.

3.1. Вывод символьных строк

Для знающих основы использования Си-подобных языков программирования первой программой, описанной в классическом труде [19], должна быть такая, которая печатает (выводит на экран) сообщение «*Hello, world!*». Ее исходный код на Питоне будет содержать только один оператор

print(«Hello, world».

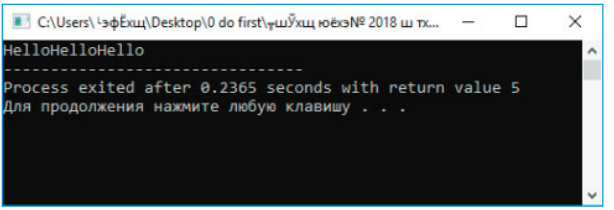
Для запуска такого кода надо сохранить его, выбрав в меню команду *File–Save* (этого же можно достичь комбинациями клавиш <Ctrl+S>), а затем выполнить – *Run–Run Module*.

В упомянутом труде Б. Кернигана и Д. Ритчи [19] после вывода сообщения переход на новую строку не осуществляется. Если выполнить такую программу на Си:

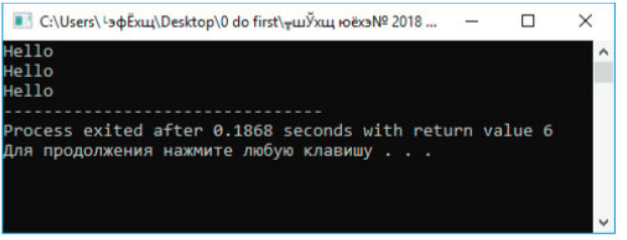
void main()

```
{
    printf(«Hello»);
    printf(«Hello»);
    printf(«Hello»);
}
```

вывод осуществится в одну строку.



Для перевода очередного вывода в следующую строку надо добавить команду printf(«Hello\n»); или printf(«\nHello»);.



В Питоне оператор print() переводит следующий вывод в новую строку. Пропустить строку можно таким оператором

print(«\n»).

Теперь разобьем выводимый текст (являющейся символьной строкой) на две части. Первая – это символы « Hello, «, а вторая – «world!». В языках программирования традиционно символьные строки заключаются в двойные кавычки. В Питоне для этого используются как двойные, так и одинарные кавычки. Выведем сообщение, предварительно соединив две заданные заранее символьные строки. В рассмотренных далее примерах показано равноправие двойных и одинарных кавычек при формировании символьных констант. В первом операторе переменная инициируется одинарными кавычками, а во втором – двойными.

```
s1 = ‘Hello, ‘
s2 = “world!”
```

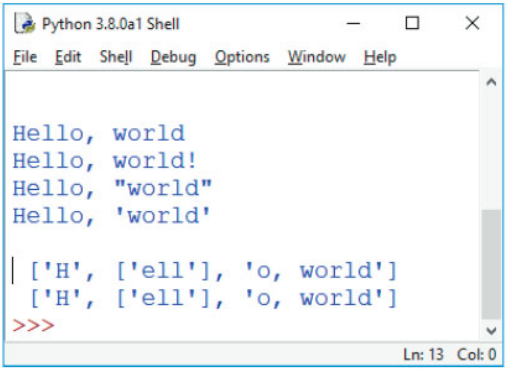
Далее переменные объединяются операцией «+», и полученное значение выводится на экран.

```
S12 = s1 + s2
print(S12)
```

В двух следующих примерах показано, что один тип кавычек может располагаться внутри других. И в третьем – интерпретация двойных и одинарных кавычек в списке (*list*).

```
S0 = ‘Hello, “world”’
print(S0)
S0 = “Hello, ‘world’”
print(S0)
listS = [“H”, [“ell”], “o”, “ world”]
print(“\n”, listS)
listS = [“H”, [‘ell’], “o, world”]
print(“”, listS)
listS = r’H’, [“ell”], ‘o’, “ world”
print(« », listS)
```

Рассмотрим скриншот с результатами приведенных фрагментов программы. Обратим внимание на то, что в списке выводятся только одиночные кавычки, хотя в его определении использованы и одиночные, и двойные.



Отметим, что перед выводом строк их содержимое можно форматировать. Для этого в Питоне имеются мощные средства,

которые опишем подробнее после изучения переменных разных типов.

3.2. Ввод данных в программу

Теперь узнаем, как вводить данные в программу, то есть задавать значения переменных программы извне. В первом фрагменте сначала запрашивается имя, а затем выводится адресное приветствие.

```
name = input(«Как Вас зовут? “)
print(“Привет,”, name)
print(“Привет,”, name*2)
```

Продолжим знакомиться с тем, как вводить данные в программу с клавиатуры. В представленном фрагменте программы вводятся два значения, и если это целые числа, то сначала они соединяются как символьные данные, а затем выполняется их преобразование к виду целых, и они складываются.

```
print(‘\n’)
print(‘Input 2 int’)
a1=input()
a2=input()
print(‘     a1 + a2: ‘,(a1+a2))
a1=int(a1)
a2=int(a2)
s=a1+a2
print(‘ now int a1+a2:  ‘, s )
```

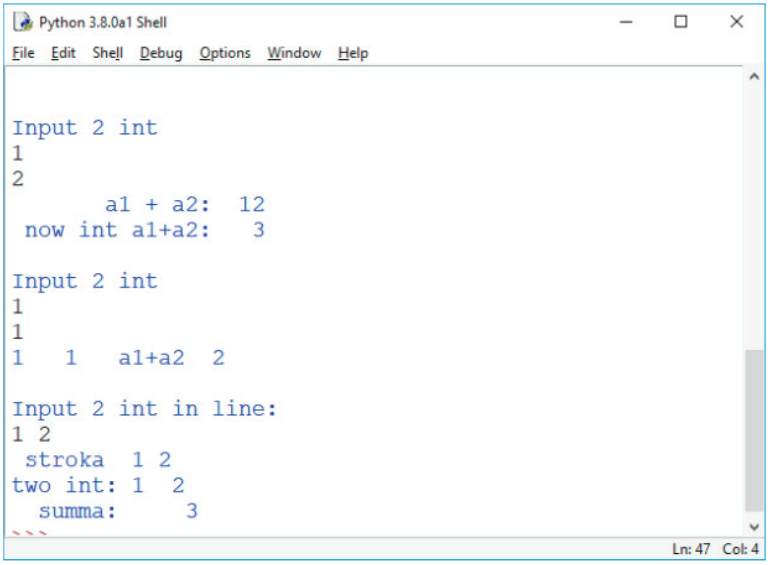
В следующем фрагменте два целых введенных значения преобразуются к целому виду сразу при вводе.

```
print(‘\nInput 2 int’)
a1=int(input())
a2=int(input())
print(a1, ‘ ‘,a2, ‘  a1+a2 ‘, a1+a2)
```

Теперь покажем, как вводить несколько целых значений с одной строке. Для этого введенный набор символов разделяется на части функций *split*().

```
print(‘\nInput 2 int in line:’)
s=input()
s=str(s)
```

```
print(‘ stroka ‘,s)
sa=s.split(’ ’)
if ( len(sa) == 2):
  print(’two int:’,sa[0],’’, sa[1])
  print(’  summa:   ’,int(sa[0])+int(sa[1]) )
```

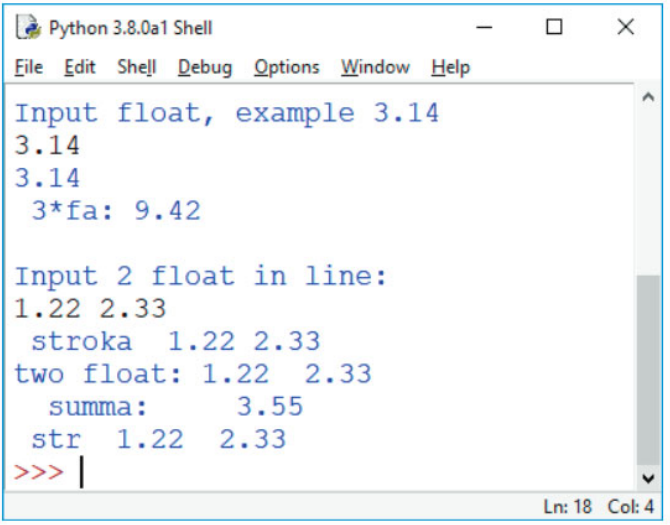


Здесь покажем, как вводить вещественные числа. Напомним, что при их вводе целая часть от дробной отделяется точкой, так же как и при написании вещественных констант.

```
print(‘Input float, example 3.14’)
f = input()
fa=float(f)
print(fa)
fa=fa*3
print(‘ 3*fa:’,fa)
print(‘\nInput 2 float in line:’)
sf=input(); sf=str(sf)
print(‘ stroka ‘,sf)
saf=sf.split(‘ ‘)
if ( len(saf) == 2):
```

```
    print('two float:',saf[0],'', saf[1])
    print(' summa:  ',float(saf[0])+float(saf[1]) )
  sf0=float(saf[0])
  sf1=float(saf[1])
  strsf=str(sf0)
  strsf=strsf+" "+str(sf1)
  print(' str ',strsf)
```

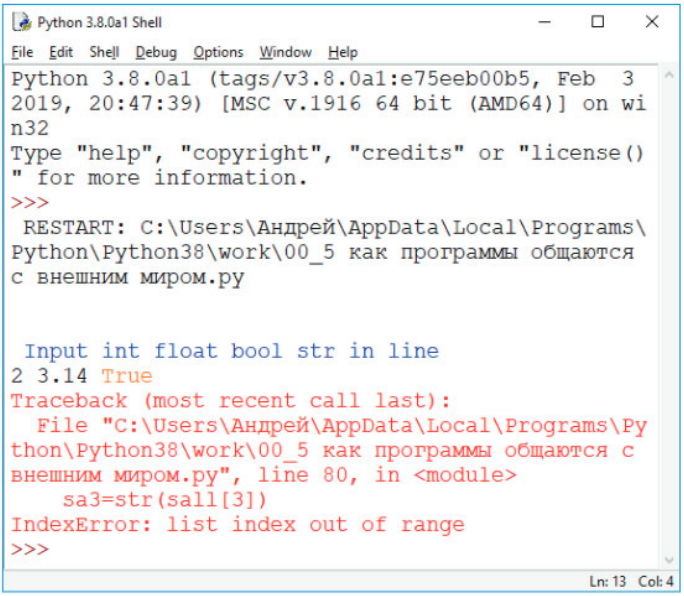
Результат работы такого фрагмента исходного приведен в следующем скриншоте.



Покажем, как в одной строке ввести данные разных типов. Вводятся целое и вещественное числа и логическое значение. И в заключение – символьное данное.

```
  print('Input 1)int 2)float 3)bool
  allType=input()
  sall=allType.split(' ')
  sa0=int(sall[0])
  sa1=float(sall[1])
  sa2=bool(sall[1])
  sa3=str(sall[3])
  print(' all in one ',sall)
```

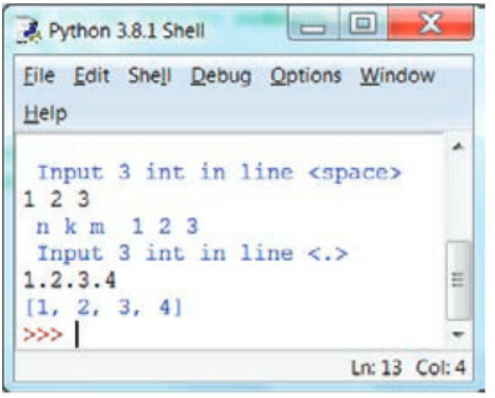
Если введено только 3 значения, выводится сообщение об ошибке.



3.3. Использование map

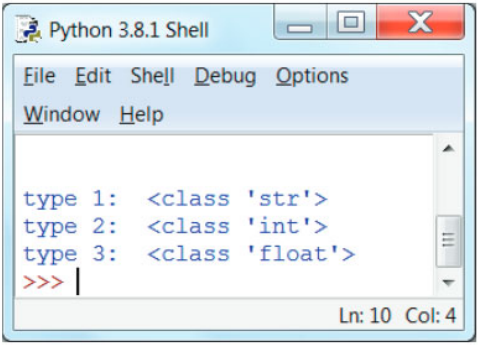
Функция *map* в Питоне применяет функцию к каждому элементу списка. Например, имеется список символьных значений, а их надо преобразовать в другой тип. Но в первом примере функция *map* применяется к трем переменным, каждая из которых (в отличие от списка) имеет только одно значение.

```
# Покажем как использовать map
print(‘ Input 3 int in line <space>’)
n,k,m=map(int, input().split())
print(‘ n k m ‘, n,k,m)
# Можно проще – список
print(‘ Input 3 int in line <.>’)
s=input().split(‘.’)
a=list(map(int,s))
print(a)
```



Во втором примере создается список символьных значений, каждый из которых есть цифра. Далее этот список преобразуется так, что его элементы – целые числа, а затем последний преобразуется в вещественный тип.

```
st1=['1', '2', '3']
print('type 1: ', type(list1[0]) )
list2=list(map(int,list1))
print('type 2: ', type(list2[0]) )
list3=list(map(float,list2))
print('type 3: ', type(list3[0]) )
```

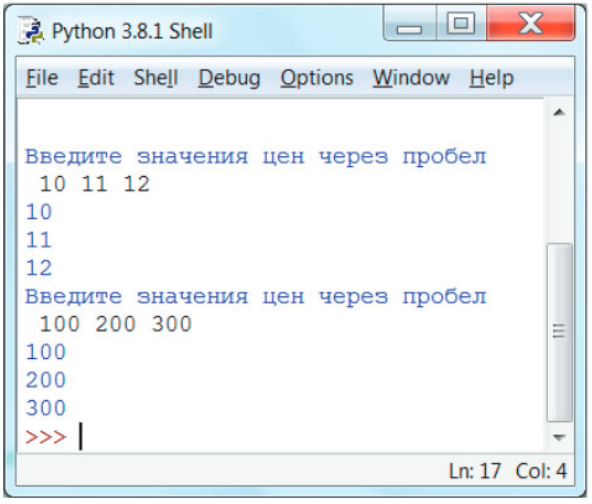


Приведем еще примеры использования функции *map* для ввода нескольких значений с клавиатуры. Эти примеры размещены на сайте по адресу – www.cyberforum.ru/python-beginners/

thread2023600.html. Сначала – два примера, в которых требуемые действия записываются в одну строку. Они показывают, что Питон позволяет использовать самые разнообразные конструкции.

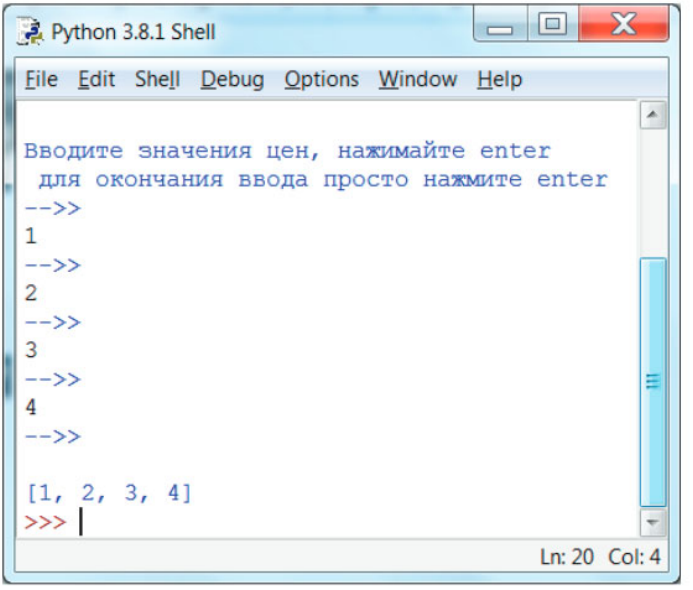
rices = list(map(int,input('Введите значения цен через пробел ').split()))

rices = [int(i) for i in input('Введите значения цен через пробел ').split()]



Приведем и еще один пример с упомянутого источника Интернет (cyberforum.ru).

```
print('Вводите значения цен, нажимайте enter')
print(' для окончания ввода просто нажмите enter')
a = int(input('-->> \n'))
rices = []
while True:
  try:
    rices.append(a)
    a = int(input('-->> \n'))
  except:
    break
print(rices)
```



Скажем несколько слов о взаимодействии переменных оболочки и программного модуля. Были выполнены следующие действия.

1. В оболочке ведена переменная присваиванием (a=5).
2. Вывел имя (a – получил, естественно, значение 5).
3. Открыл новый файл.
4. Пытаюсь сразу вывести значение a (print(a))

- ОШИБКА: NameError: name 'a' is not defined

Теперь наоборот. Добавил в модуле такие операторы:

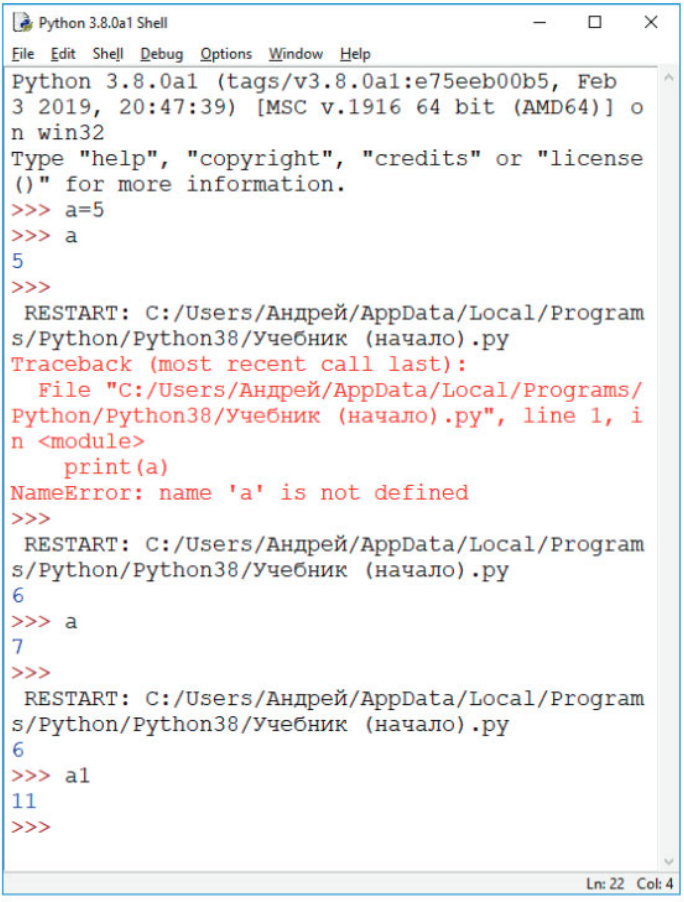
a=6

print(a)

a=7

и запустил.

В оболочке, естественно, выводится 6, но после указания имени a в оболочке, выводится 7. Повторим еще раз. Введем переменную a1 в модуле и присвоим ей значение 11, выполним модуль. Получим 11. Если вывести a1 в оболочке, получим также 11.



Экзаменационные темы

1. Меню редактора текстов **Python.**
2. Вывод символьных данных в Питоне.
3. Ввод данных разного типа в программу на Питоне.
4. Использование функции *map.*
5. Взаимодействие переменных оболочки и программного модуля.

4. ЧИСЛОВЫЕ ДАННЫЕ В ПИТОНЕ

В данной части учебника рассматривается средства языка *Python*, которые уже частично рассматривались ранее. Приведенные примеры дополняют материал, приведенный ранее.

4.1. Целые числа

Начнем с целых чисел. Следующий пример показывает основные операции над ними. Кроме сложения, вычитания и умножения определены еще и такие действия:

– остаток от деления целых;
– деление;
– целочисленное деление.

```
i1 = 7; i2 = 3
print(' i1 7  ','i2 3  ' )
print(' i1%i2  ', (i1%i2) , ' ' )   # Остаток от деления целых
print(' i1/i2  ', (i1/i2) , ' ' )     # Деление
print(' i1//i2 ', (i1//i2), ' ' )     # Целочисленное деление
iChar1='7'; iChar2='3'
print(' iChar1+iChar2 ', (iChar1+iChar2), ' ' )   # Сложение
```

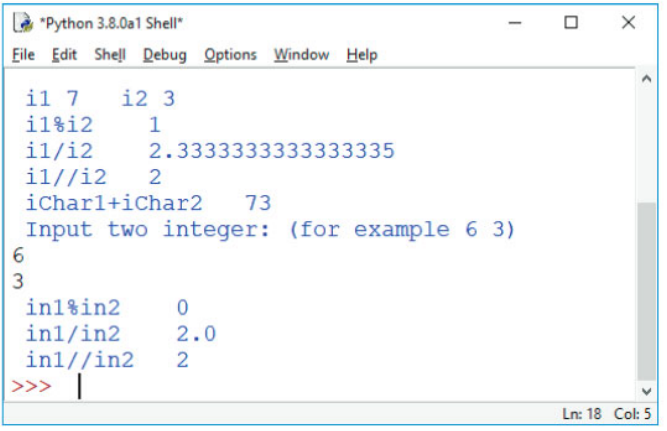
Последний вывод показывает для сравнения, как выполняется операция «+» для символьных данных.

Следующие примеры повторяют предыдущие, но исходные данные для них вводятся с клавиатуры.

```
print(' Input two integer: (for example 6 3)')
# NO: in1 = input(); in2 = input()
in1 = int(input()); in2 = int(input())
print(' in1 in2  ', (in1%in2) , ' ' )
print(' in1/in2  ', (in1/in2) , ' ' )
print(, in1//in2  ,, (in1//in2), ,  , )
```

ЗАМЕЧАНИЕ. В Питоне, как и других языках, можно в исходный текст добавлять комментарии. Они бывают двух типов. Строчный комментарий фиксируется одним знаком #. А многострочные комментарии начинаются тремя одинарными кавычками и заканчиваются ими же.

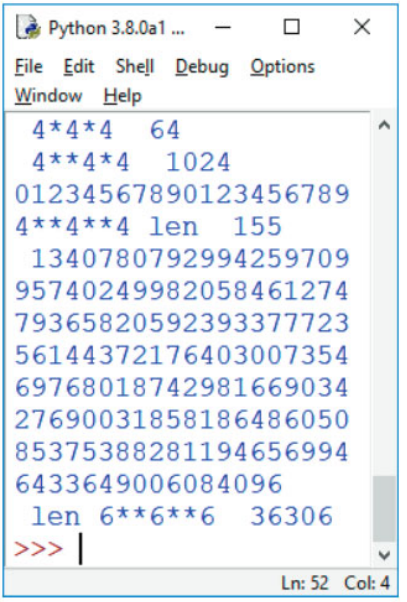
```
# Это комментарий, записанный до конца строки
''' Многострочный комментарий
может занимать несколько строк
'''
```



В Питоне данные целого типа могут принимать большие значения. Это отличает данный язык от многих других, где память, выделяемая для переменных и констант целого типа, статична. При таком подходе максимальное значение весьма невелико. К примеру, если отвести под элемент данных целого типа 32 бит, то его максимальное значение будет 4 294 967 295, то есть около 4 миллиардов, или около 4ё109.

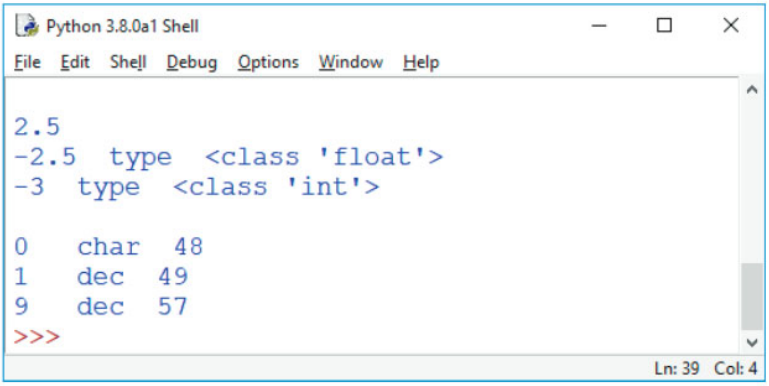
Следующие примеры показывают, что метод выделения памяти для целых значений отличается от описанного выше. Это позволяет формировать целое значение с 451 цифрами и даже 36 306. Здесь использована операция возведения в степень (**).

```
# Максимальное целое
print(‘ 4*4*4 ‘, 4*4*4)
print(’ 4**4*4 ’, 4**4*4)
print(’01234567890123456789’)
Chislo=4**4**4
strCh = str(Chislo)
lenCh = len(strCh)
print(’4**4**4 len ’, lenCh, ’\n’, 4**4**4)
Chislo = 6**6**6
strCh = str(Chislo)
lenCh = len(strCh)
print(‘ len 6**6**6 ‘, lenCh)
# len 6**6**6  = 36306 символов
```



Ранее приводились примеры использования функции *type*. Здесь показывается, как фиксируется тип для переменной, которая получается делением целочисленного отрицательного числа на положительное. Также показано, что результатом деления нацело (−5) на 2 будет −3, а не −2, как принято в арифметике.

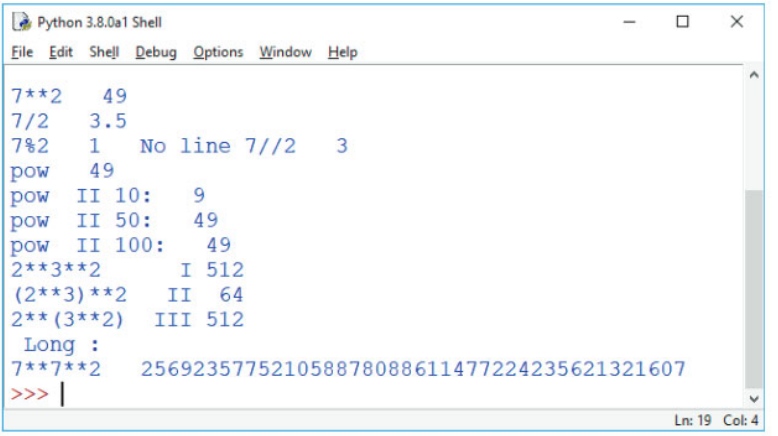
```
a=5/2
print(a)
a_minus1 =(-5)/2
print(a_minus1, ' type ', type(a_minus1) )
a_minus2 =(-5)//2
print(a_minus2, ' type ', type(a_minus2) )
print()
s0char='0'
print(s0char, '  char ', ord(s0char))
s1=19
d=str(s1)
print(d[0], '  dec ', ord(str(d[0])) )
print(d[1], '  dec ', ord(str(d[1])) )
```

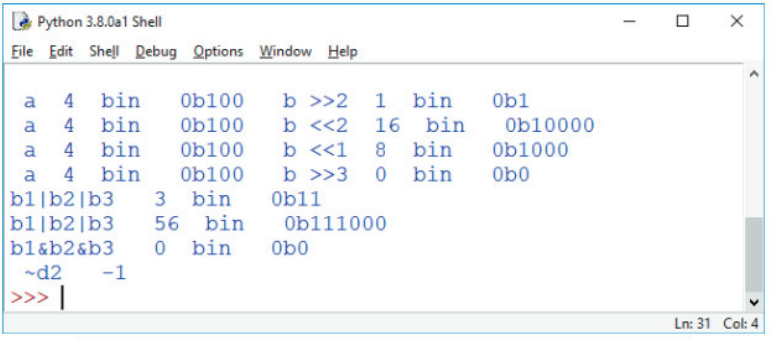


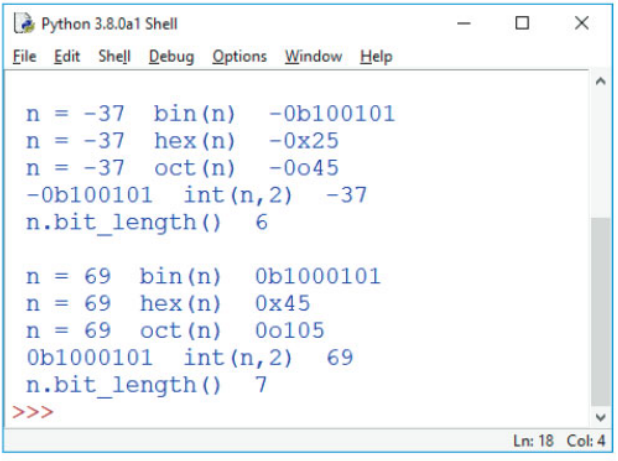
Три последние строки показывают использование функции *ord*, выводящей код символа.

Следующий фрагмент исходного текста демонстрирует использование функции возведения в степень (*pow*). Функция *pow* имеет два обязательных параметра: возводимое число и степень. Третий параметр является необязательным и определяет, что число, возвращаемое как результат, является остатком деления от него. Так, результатом выполнения операции «остаток от деления на 10» числа 49 будет 9. А остаток от деления 49 на 50 и 100 будет это число.

```
print(«7**2 «, 7**2)
print("7/2 ", 7/2)
print("7%2 ", 7%2,end=' ')
print(" No line 7//2 ", 7//2)
print("pow ", pow(7,2))
print("pow II 10: ", pow(7,2,10))
print("pow II 50: ", pow(7,2,50))
print("pow II 100: ", pow(7,2,100))
print("2**3**2 I", 2**3**2)
print("(2**3)**2 II ", (2**3)**2)
print("2**(3**2) III", 2**(3**2) )
print(' Long :')
print(«7**7**2 «, 7**7**2 )
```





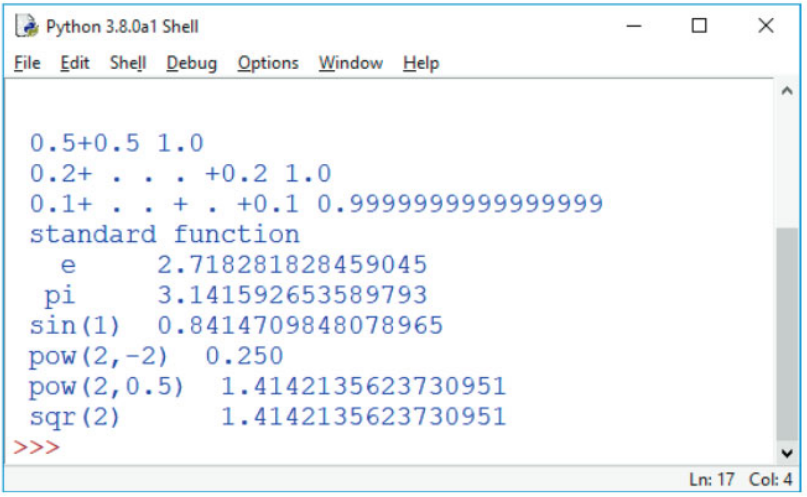


Перейдем к рассмотрению вещественных чисел.

4.2. Вещественные числа

В рассматриваемом языке поддерживается три типа чисел. Целые уже рассмотрены. Комплексные – в следующем разделе. А теперь обратимся к вещественным числам. Любые данные в компьютере сохраняются как набор 0 и 1, то есть в двоичном представлении. Это приводит к некоторым проблемам при их использовании. Для вещественных чисел это округление при выполнении действий, что может привести к нежелательной потере точности. Во многих источниках описан пример, приведенный далее. Но следует отметить, что значение двух чисел: *e* и пи (ϖ) выводится довольно точно при использовании встроенных функций. Для использования математических функций в текст программы надо добавить оператор подключения модуля ***math*** стандартной библиотеки.

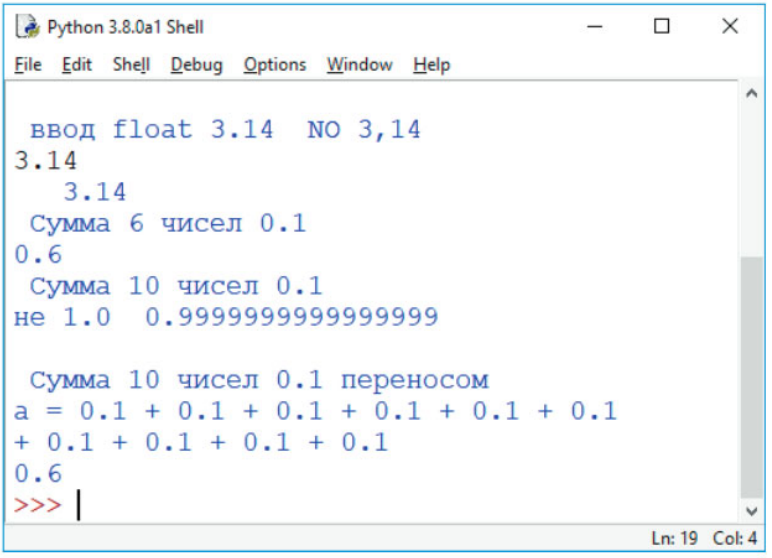
```
# Python world, стр.34

r=0.5+0.5
print(‘ 0.5+0.5’, r)

r=0.2+0.2+0.2+0.2+0.2
print(‘ 0.2+ . . . +0.2’, r)

r=0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1
print(‘ 0.1+ . . + . +0.1’, r)

import math
print(‘ standard function ‘)
print(‘  e   ‘, math.e )
print(‘  pi   ‘, math.pi )
#  print(‘  pi   ‘, math.pi ) Syntax Error: indetected indent
print(‘ sin(1) ‘, math.sin(1) )
print(‘ pow(2,-2) {0:6.3f} ‘.format( math.pow(2,-2) ) )
print(‘ pow(2,0.5) ‘, math.pow(2,0.5) )
print(‘ sqr(2)    ‘, math.sqrt(2) )
```



Следующий пример демонстрирует, как можно целочисленное значение ввести с клавиатуры. При этом целая часть от дробной отделяется точкой, а не запятой. Показано, что в Питоне есть только один вещественный тип *float*. В некоторых языках программирования есть еще и второй – *double*. Также показано, как можно записать оператор не в одну, а несколько строк. В комментарии приведен неверный пример.

```
print(‘ ввод float 3.14  NO 3,14’)
af = float(input())
print(‘  ‘,af)
# NO double  ad = double(input())

print(‘ Сумма 6 чисел 0.1’)
a = 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1
print( a)

print(‘ Сумма 10 чисел 0.1’)
a = 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1
print(‘не 1.0 ‘, a)

# Неверный перенос
# a = 0.1 + 0.1 + 0.1 + 0.1 + 0.1 +
```

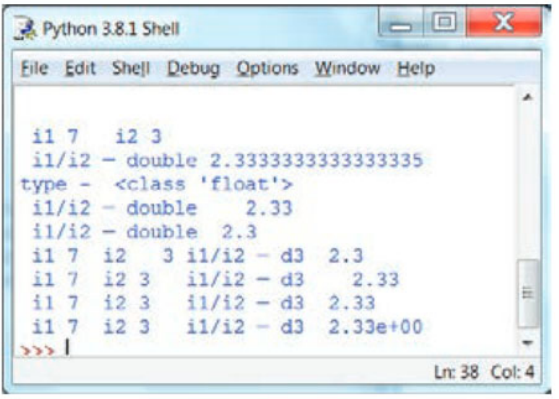
```
# + 0.1 + 0.1 + 0.1 + 0.1 + 0.1
print()
print(‘ Сумма 10 чисел 0.1 переносом’)
print(‘a = 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1’)
print(‘+ 0.1 + 0.1 + 0.1 + 0.1’)
a = 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1
+ 0.1 + 0.1 + 0.1 + 0.1
print(a)
```



Заключительный пример раздела о вещественных числах показывает, что при обычном делении чисел формируется вещественное значение и приводятся конструкции, форматирующие значение перед выводом на экран. Для вывода вещественного числа применяются два форматных кода: *f* и *e*. Первый фиксирует общее количество знаков в числе и его дробной части. Второй выводит число в экспоненциальной форме.

```
i1 = 7; i2 = 3; print(’ i1 7  i2 3’); d3= i1/i2
print(’ i1 %d  i2 %d  i1/i2 — d3  %4.2e’ %(i1,i2,d3) )  # NOT f in format
print(’ i1 {0:d}  i2 {1:d} i1/i2 — d3  {2:3.2}’.format( i1,i2,d3) )
```

```
print(' i1 %d  i2 %d i1/i2 — d3  %6.2f' %(i1,i2,d3) )
print(' i1 %d  i2 %d i1/i2 — d3  %3.2f' %(i1,i2,d3) )
print(' i1 %d  i2 %d   i1/i2 — d3  %4.2e' %(i1,i2,d3) )
```



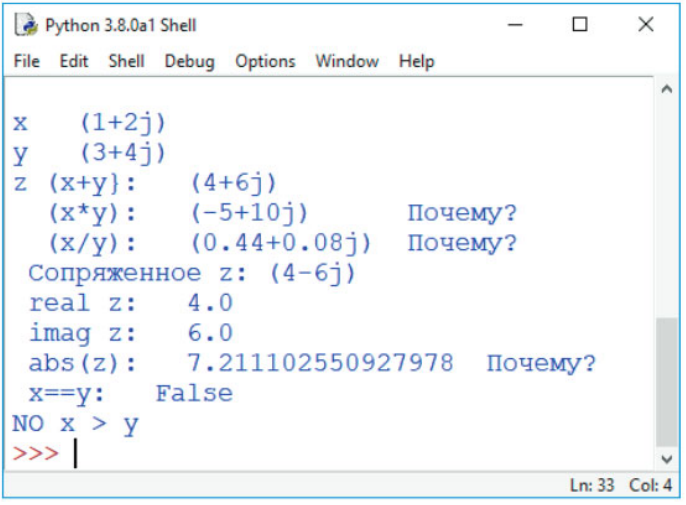
Перейдем к описанию третьего типа числовых данных.

4.3. Комплексные числа

Питон поддерживает комплексные числа и операции над ними. Они выглядят так: a+jb. Здесь a и b целые или вещественные числа. Сама по себе мнимая единица не может использоваться, но можно записать ее так 1j. В приведенном фрагменте кода Питона даются примеры способов задания и выполнения действий (сложения, вычитания, умножения, деления) над комплексными числами. Также показано, как получить действительную и мнимую части таких чисел, а еще сопряженное число. Еще продемонстрировано, как комплексные числа можно сравнивать, а как нельзя.

```
x = complex(1, 2)
print('x  ',x)

y = complex(3.0, 4)
print('y  ',y)

z = x + y
print(,z (x+y}:  ,, z)
```

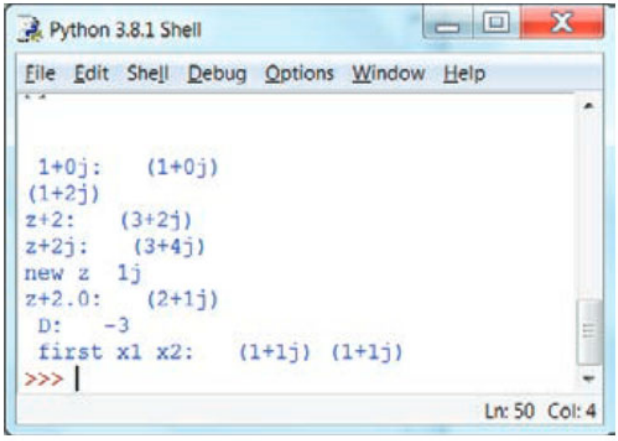
```
zm = x * y
print(‘  (x*y): ‘,zm, ‘ )

zd = x / y
print(‘  (x/y): ‘,zd, ‘ Почему?’)
print(‘ Сопряженное z:’, z.conjugate()) # Сопряженное число
print(‘ real z: ‘,z.real) # Действительная часть
print(‘ imag z: ‘,z.imag) # Мнимая часть
print(, abs(z): ,, abs(z), , Почему?‘) # Модуль числа
print(‘ x==y: ‘, x == y) # Комплексные числа можно сравнивать так
print(‘NO x > y ‘) # Комплексные числа НЕЛЬЗЯ сравнивать так
```



В последнем фрагменте расширяются знания об использовании комплексных чисел. Сначала показано, как к комплексному числу и мнимой единице (частный случай) добавляется вещественное число.

```
# Комплексные числа (продолжение)
print(‘ 1+0j: ‘, 1+0j)

z=1+2j
print(z)
```

```
z=z+2.0
print(,z+2:  ,,z)

z=z+2.0j
print(,z+2j:  ,,z)

z=1j
print('new z ', z)

z=z+2.0
print(,z+2.0:  ,,z)

import math
a=1; b=-1; c=1
D=b*b-4*a*c
print(' D:  ',D)
# NO NO NO (D<0)  sD = math.sqrt(D), if D<0
#              print('sD:  ',sD)

if D < 0 :
   D=-D
# NO NO NO Read only attribute
'
x1=1+1j
x2=1+1j
print(' first x1 x2:  ', x1, x2)
```



В следующем разделе рассмотрим более подробно операторы языка программирования Питон.

Экзаменационные темы

1. Целые числа и операции над ними.
2. Вещественные числа и операции над ними.
3. Комплексные числа и операции над ними.
4. Комментарии в Питоне.

5. ДВЕ ОСНОВНЫЕ АЛГОРИТМИЧЕСКИЕ КОНСТРУКЦИИ В ПРОГРАММИРОВАНИИ

В каждом языке программирования имеется набор ключевых слов, которые во многом определяют его функциональность. Список ключевых слов получим, выполнив в оболочке такую команду.

keyword.kwlist

Но предварительно надо сделать доступным модуль *keyword* так:

import keyword

Разные языки программирования содержат операцию, которая фиксируется не ключевым словом, а одним символом – знаком равенства. Последний определяет присваивание. В Питоне оператор присваивания выполняет, кроме обычной для многих языков программирования функции смены значения переменной, еще и функцию определения ее типа. Это уже было продемонстрировано ранее (см. п. 2.5).

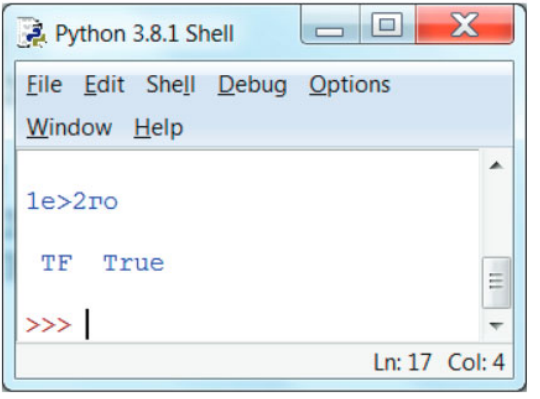
5.1. Условный оператор (подробнее)

Теперь перейдем к более подробному рассмотрению возможностей условной конструкции. Она начинается с ключевого слова *if*, после которого записывается условие. Последнее является выражением, которое приобретает только два возможных значения: истина (ключевое слово *True*) или ложь (ключевое слово *False*). Таким образом, условие является выражением логического типа. Для переменных такого типа в Питоне есть два значения *True* (истина) и *False* (ложь). Это является известным местом для многих языков программирования.

В примере, рассмотренном далее, эти значения выводятся в зависимости от того, какая из двух переменных имеет большее значение. Точнее, если больше первая, то выводится *True*, в противном случае – *False*.

```
f=2; t=1
if f>t :
  TF=True
  print('1e>2ro\n', )
```

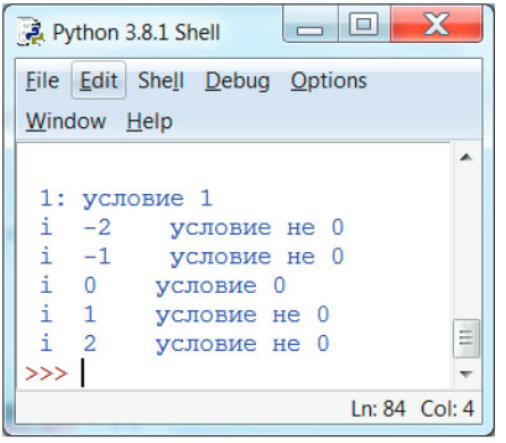
```
else:
  TF=True
  print('1e>2ro\n', )
#TF=a>b
print(' TF ',TF, '\n')
```



На самом деле для решения данной задачи можно просто записать один оператор присваивания, как приведено в комментариях в тексте фрагмента – TF=a>b.

В Питоне условием могут быть константы. Но тогда действия, записанные после строки с *if*, либо выполняются всегда (if True :), либо не выполняются никогда (if False).

```
if ( 1 ): # Можно использовать скобки
  print(' 1: условие 1')
else:
  print(' 1: условие не 1')

for i in range(-2,3):
  if ( i ):
    print(' i ',i, '  условие не 0')
  else:
    print(' i ',i, '  условие 0')
```



Приведенный фрагмент показывает, что в условиях любое число не 0 равносильно константе *True*, а 0 и другие значения – *False*.

Классический оператор *if* реализует конструкцию, называемую условной. При этом есть две ее разновидности: «Разветвление» и «Обход». При реализации первой в зависимости от значения условия, если оно истинно, выполняется одна группа действий, а если оно ложно – другая. При «Обходе» ситуация такая. Если условие истинно, выполняется группа действий, если ложно – не выполняется ничего. Этот материал доступен в любом школьном учебнике по «Информатике». Приведем фрагмент программы, в котором задача нахождения максимального значения из двух чисел реализована с помощью конструкции «Обход». При этом первое из двух чисел сначала присваивается максимальному значению. А затем оно сравнивается со вторым значением и, если нужно, заменяется на его значение.

Питон, как и другие языки программирования, предлагает более сложную условную конструкцию. Например, в задаче сравнения двух чисел выделяются три случая:

– первое число больше второго;

– оба числа равны;

– второе число больше первого.

Такую задачу можно реализовать разными способами. В первом используется вложенное разветвление.

```
a=2; b=2
print('a= ',a, '  b= ',b)
if a == b:
  print("a=b")
else:
  if a>b:
    print('a>b')
  else:
    print('a<b')
```

Во второй реализации используется конструкция *if–elif–else*.

```
a=3; b=2
if a == b:
  print(«a=b»)
elif a>b:
    print(«a>b»)
else
  print(«a<b»)
```

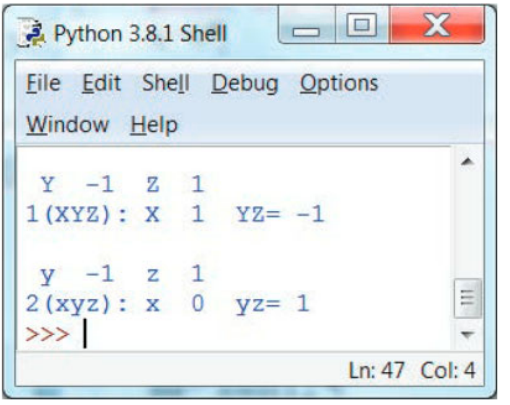
Для присваивания переменной двух различных значений в зависимости от некоторых условий в Питоне имеется специальная конструкция. В приведенном примере переменная *YZ* получает значение в зависимости от значения *X*.

```
Y=-1; Z=1
print(' Y ',Y, ' Z ',Z)
X=1
if X:
  YZ=Y
else:
  YZ=Z
print('1(XYZ): X ',X, ' YZ=', YZ)
```

В этом примере используется трехместная конструкция *if/else*.

```
y=-1; z=1
print('\n y ',y, ' z ',z)
x=0
yz = y if x else z
print('2(xyz): x ',x, ' yz=', yz)
```

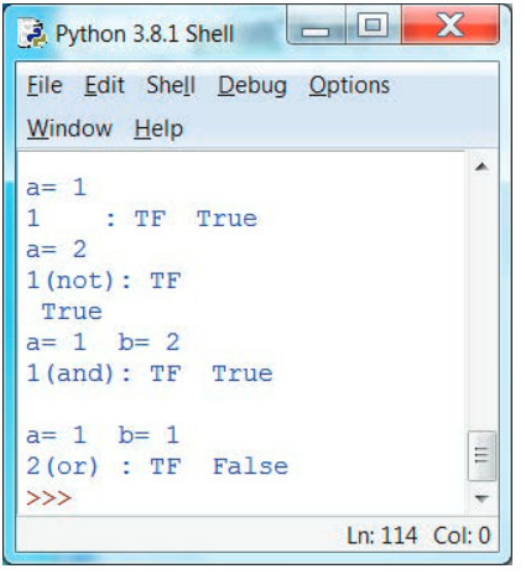
Приведем результаты обоих фрагментов.



Используемые ранее условия являются простыми. Их них могут образовываться сложные. Для этого используются логические операторы *not*, *and* и *or*. Первый изменяет истинностное значение на противоположное. Логический оператор *and* связывает два условия и будет истинным, если истины оба. Логический оператор *or* связывает два условия и будет истинным, если истинно хотя бы одно.

```
a=1
print('a=',a)
if a==1:
  TF= (a==1)
  print('1   : TF ', TF)
a=2
print('a=',a)
if not a==1:
  TF= not(a==1)
  print('1(not): TF \n', TF)

a=1; b=2
print('a=',a, ' b=',b)
if a==1 and b==2:
  TF=(a==1) and (b==2)
  print('1(and): TF ', TF)
```

```
a=1; b=1
print('\na=',a, ' b=',b)
if a==1 or b==2:
   TF=(a==1) and (b==2)
   print('2(or) : TF ', TF)
```



В Питоне сложные условия могут связывать более двух простых условий.

```
a=1; b=2; c=3; d=4
if a==1 and b==2 and c==3 and d==4:
   TF = a==1 and b==2 and c==3 and d==4
   print('3(and) : TF ', TF)
```

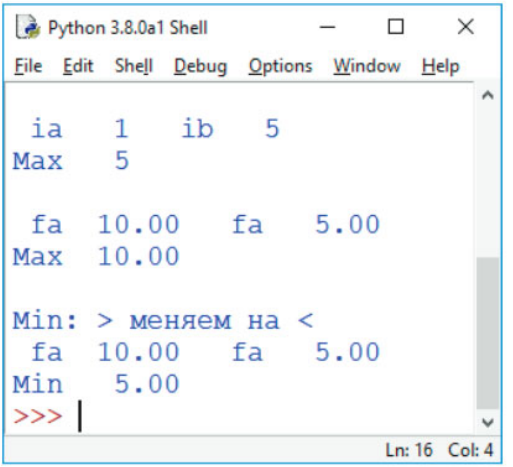
В следующем примере сложное условие используется для проверки: является ли символ цифрой. Сначала строке присваивается набор символов. Далее в цикле выделяется каждый из них, и он проверяется, будет ли его код в интервале от 48 до 58.

```
str='a9sdf12'
numStr=''
for s1 in str:
   print(s1, ' ' ord(s1) )
```

```
    # NO int(s1)!!!  if ( (int(s1)>47) and (int(s1)<59) ):
    #    if  47 < (int(s1)) <59
    #    if ( (ord(s1)>47) and (ord(s1)<59) ):
      if  47 < (ord(s1)) <59:
        print(s1)
        numStr=numStr+s1
    print(numStr)
```

Приведем примеры поиска максимального и минимального значений, в которых используется форматирование выводимых значений.

```
    # Maximum 1
    ia = 1; ib = 5;
    if (ia > ib):
      iMax = ia;
    else:
      iMax = ib;
    # print("Max {0}".format(iMax));
    # Можно
    print(" ia {0:3d}   ib {1:3d}".format(ia,ib));
    print(„Max {0:3d}".format(iMax));

    # Maximum 2
    print()
    fa = 10.0; fb = 5.0;
    fMax=fa
    if (fb > fMax):
      fMax = fb
    print(" fa %6.2f   fa %6.2f" %(fa,fb));
    print("Max %6.2f" %(fMax));

    # Minimum 3
    print('\nMin: > меняем на <')
    fMin=fa
    if (fb < fMin):
      fMin = fb
    print(" fa %6.2f   fa %6.2f" %(fa,fb));
    print("Min %6.2f" %(fMin));
```



В следующем примере условная конструкция используется для определения: принадлежит ли заданная точка одному из двух заданных интервалов (рис. 5.1). Значения чисел, определяющие интервалы, следующие: $a = -2.0$, $b = 0.0$, $c = 2.0$ и $d = 5.0$.

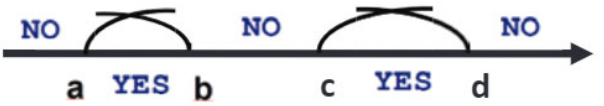


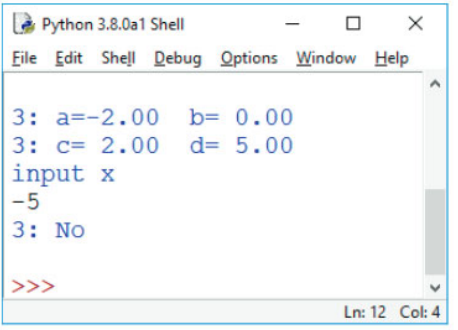
Рис. 5.1. Принадлежность точки любому из двух заданных интервалов

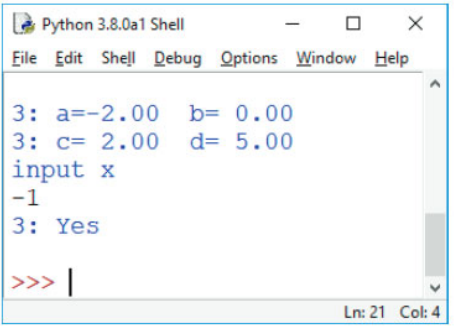
Программа, решающая поставленную задачу, может быть такой:

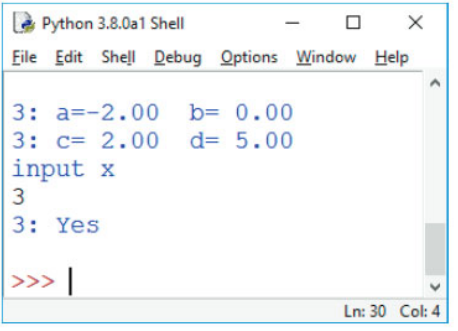
```
a = -2.0;  b = 0.0;  c = 2.0;  d = 5.0;
print("3: a={0:5.2f}  b={1:5.2f}".format(a, b))
print("3: c={0:5.2f}  d={1:5.2f}".format(c, d))

print('input x')
x = float(input())
TF1 = (x >= a) and (x <= b)
TF2 = (x >= c) and (x <= d)
if (TF1 or TF2):
```

```
    print("3: Yes\n")
else:
    print(«3: No\n»)
```

Приведем несколько результатов работы программы с разными значениями точки x на оси.







А сейчас рассмотрим, как в Питоне реализуется циклическая конструкция. В отличии от других языков в Питоне их две формы.

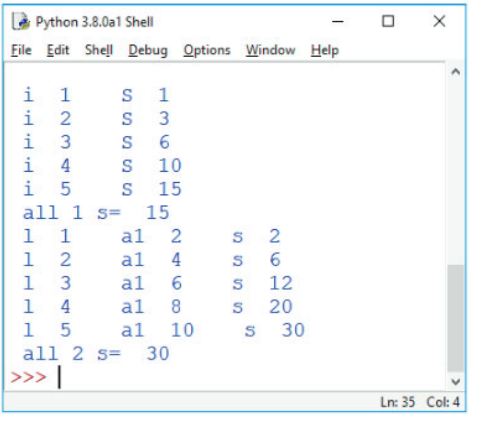
5.2. Циклы for и while

Теперь расширим знания об использовании циклических конструкций. Напомним, что в Питоне их две: *while* и *for*. В приведенном далее примере рассмотрим первую. Решается задача вычисления суммы пяти членов натурального ряда.

```
# 1+2+3+4+5
a=1; s=0
while a<=5:
  s=s+a
  print(' i ',a, '  S ',s)
  a=a+1
print(' all 1 s= ',s)
```

Используя предыдущий алгоритм, можно решить другую задачу, заменив лишь способ вычисления очередного слагаемого. Вычисляется сумма пяти четных значений натурального ряда, и показано, как это можно сделать, используя *for*.

```
# 2+4+6+8+10
l=1; s=0
# while l<=5:
for i in range(1,6):
  a1=l*2
  s=s+a1
  print(' l ',l, '  a1 ',a1, '  s ',s)
  l=l+1
print(' all 2 s= ',s)
```

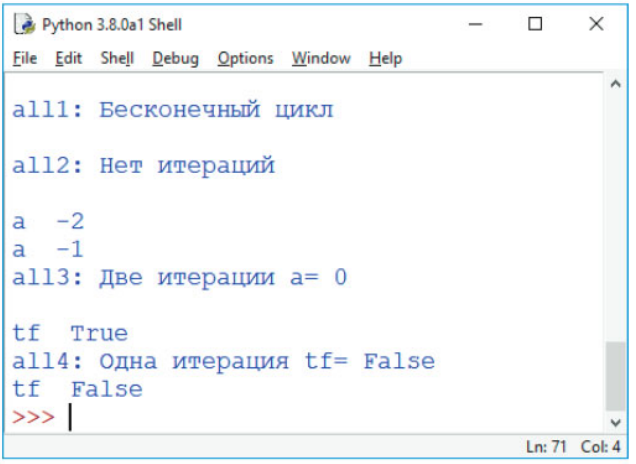


Вновь вернемся к оператору цикла *while*. После ключевого слова записывается условие, рассмотренное ранее при изучении условной конструкции. Напомним, что оно имеет два значения: *True* (истина) и *False* (ложь), но имеет числовые эквиваленты. Если условие не равно 0, цикл будет выполняться бесконечно, а если равно 0 – ни разу. Если условие изменяется от –2 до 0, цикл выполнится два раза. Это показано далее.

```
# Бесконечный цикл
#while 1:
#    print('1')
#print('all')
print('all1: Бесконечный цикл\n')
# Нет цикла
a=0
while a:
  print('a ', a)
print('all2: Нет итераций\n')
 # Цикл выполняется два раза для условия –2 и –1.
a=-2
while a:
  print('a ', a)
  a=a+1
print('all3: Две итерации a=',a,'\n')
```

Такой цикл выполнится один раз, пока условие не станет равным ложным (*False*).

```
tf=True
while tf:
  print('tf ', tf)
  tf=False
print('all4: Одна итерация tf=',tf)
print('tf ', tf)
```



Теперь продолжим рассматривать циклические структуры. Узнаем, как они реализуются оператором *for*. В первом примере создается список из трех символьных значений. Далее в цикле выводятся как значение всех элементов списка, так и их номер. Для этого используется функция *enumerate*(). Она применяется для объектов с множественными значениями (коллекциями, которые будут подробнее рассмотрены далее) и генерирует два элемента – индекс элемента и его значение.

```
sub=('1','2','3')
for i,sub in enumerate(sub):
    print(i,sub)
```

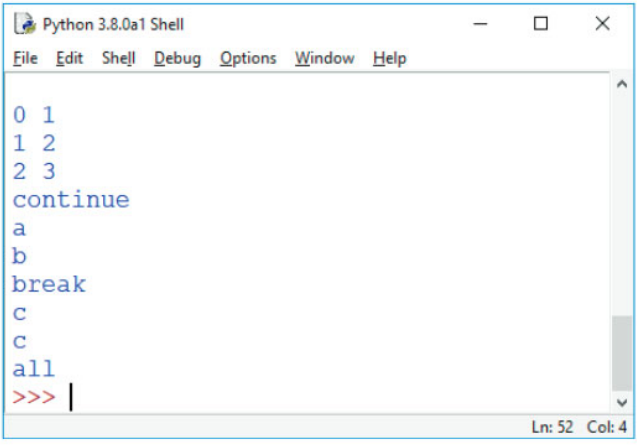
Будет выведено следующее

```
0 1
1 2
2 3
```

Теперь рассмотрим еще две языковые конструкции, которые применяются в циклах: *continue* и *break*. Первая переходит к следующей итерации цикла без выполнения операторов, записанных после *continue*. Но цикл будет продолжен, пока не наступит условие окончания цикла. В отличие от *continue*, *break* завершает цикл сразу. Следующий пример показывает это: в цикле анализируется содержимое текстовой строки: является ли очередной ее элемент символом «с».

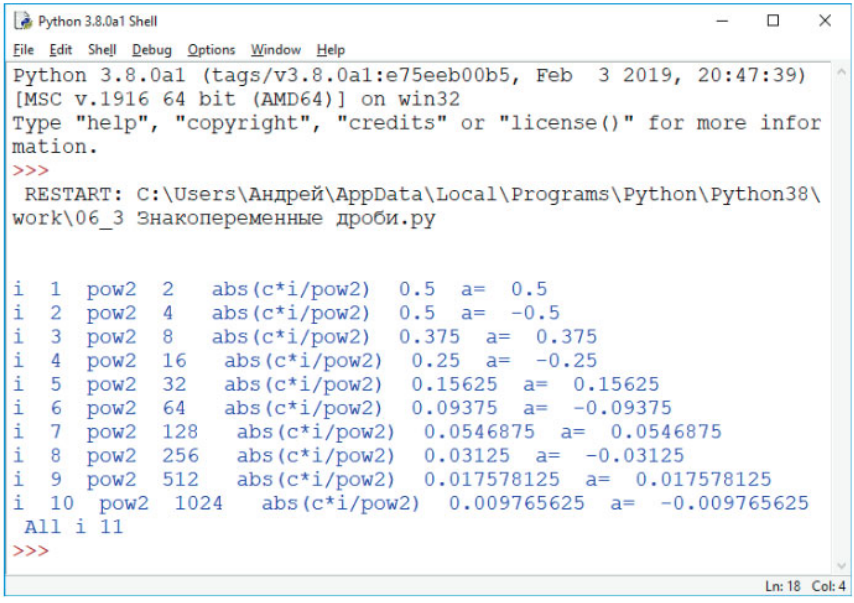
```
sub1='abccccc'
#for i,sub1 in sub1: Показано, что нужно получать два элемента.
# ValueError: not enough values to unpack (expected 2, got 1)
print('continue')
for i in sub1:
  if i == 'c':
    continue
  print(i)
#break
print('break')
sub2='ccab'
for i in sub2:
  if i !='c':
    break
  print(i)
print('all')
```

В скриншоте показаны результаты выполнения последних трех фрагментов исходного текста.



В следующем примере показано, как суммировать знакопеременную дробь. Количество слагаемых не задано. Сложения выполняются, пока очередное слагаемое не станет по модулю меньше заданного числа.

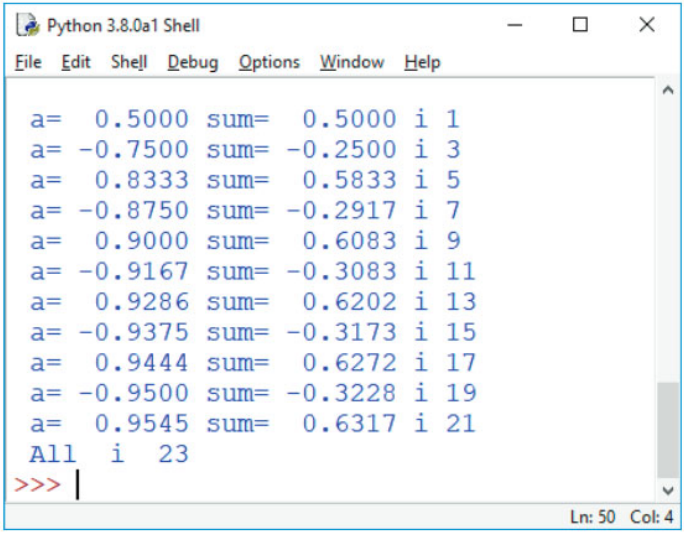
```
    # 1/2-2/4+3/8-4/16+. . .
    i=1; c=-1; Eps=0.01
    Sum = 0.0; pow2 = 1
    tf = True
    while ( tf ):
      c=-c; pow2=pow2*2
      Sum=Sum + c*i/pow2
       print('i ',i, ' pow2 ', pow2, '  abs(c*i/pow2) ', abs(c*i/pow2), '
a= ',c*i/pow2)
      if ( abs(c*i/pow2) < Eps ):
        tf = False
      i=i+1
    print(' All i', i)
```



Следующий пример похож на предыдущий. Но в нем результаты промежуточных вычислений выводятся с использованием форматирования строки.

Вычисление суммы с выводом слагаемых 1/2 +(-3/4) + 5/6 (-7/8) + . . .

```
i=1; c=-1;
Sum = 0.0
tf = True
while ( tf ):
   c=-c; a=c*i/(i+1)
   Sum=Sum + a
   print(' a= {0:7.4f} sum= {1:7.4f} i {2}'.format( c*i/(i+1), Sum,
i) )
   i=i+2
   if ( abs( a ) > 0.95 ):
      tf = False
print(' All  i ', i)
```



Экзаменационные темы

1. Формы записи условной конструкции в Питоне.
2. Простые условия в Питоне.
3. Сложные условия в Питоне.
4. Константные значения, формируемые условиями.
5. Цикл *while* в Питоне.
6. Цикл *for* в Питоне.

6. СТРОКОВЫЕ ДАННЫЕ

Наряду с числовыми данными строки имеют большое значение. Именно такой тип информации используется при «общении» человека с компьютером.

6.1. Основы работы со строками

Строковые данные в Питоне относятся к основным. Их можно создать тремя способами. Первые два – использование кавычек: двойных или одинарных. Они равноправны. Но если строка (строковое данное) начинается с двойных кавычек, то такие же кавычки должны определить ее завершение. Это же касается и одинарных кавычек. Приведенный фрагмент также показывает, что внутри одинарных кавычек можно использовать двойные и наоборот. При этом внутренние кавычки не обязательно должны быть парными.

```
print('\nравноправие кавычек')
s1 = 'Hello '
s2 = "world!"

STR = s1 + s2
print(STR)

STR = "Jack's"
print(STR)

S0='Hello, "world"'
print(S0)

S0="Hello, 'world'"
print(S0)
```

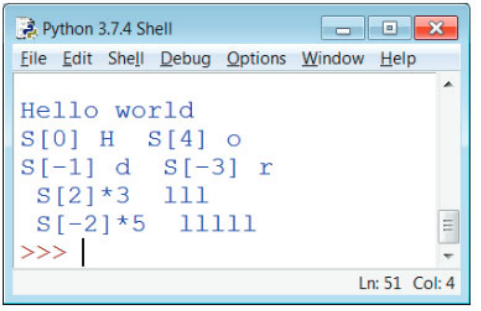
К элементу строки можно обратиться по его номеру, записываемому после имени строки в квадратных скобках.

```
S0='Hello world'

S='Hello'+" world"
print(S)
print('S[0]', S[0], ' S[4]', S[4])
print('S[-1]',S[-1],' S[-3]', S[-3])
```

Знак умножения дублирует строки. Вот примеры его использования.

```
lll = S[2]*3
```

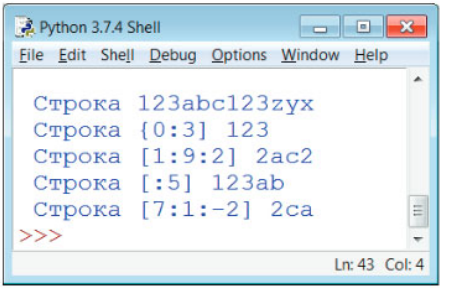
```
print(‘ S[2]*3 ‘,lll)
lllll = S[-2]*5
print(‘ S[-2]*5 ‘,lllll)
```



Еще одно действие, которое можно выполнить в Питоне со строками, – взять ее часть. Для этого после имени строки в квадратных скобках указывается:

Начальный индекс : Последний индекс : Шаг

```
Строка = ‘123abc123zyx’
print(‘ Строка’, Строка)
print(‘ Строка {0:3]’, Строка[0:3])
print(‘ Строка [1:9:2]’, Строка[1:9:2])
print(‘ Строка [:5]’, Строка[:5])
print(‘ Строка [7:1:-2]’, Строка[7:1:-2])
```



Обратим внимание, что граничные значения при отборе индексов не достигаются. Так, во второй строке отобраны символы с индексами от 0 до 2, в третьей – с номерами 1, 3, 5, 7 и в последней – с номерами 7, 5, 3.

Здесь приведем еще основные операции со строками:

– длина строки (функция len());

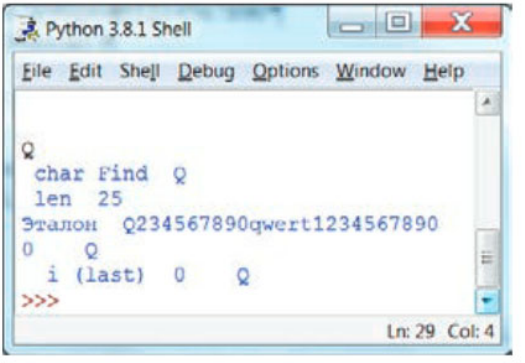
– конкатенация (сложение +);

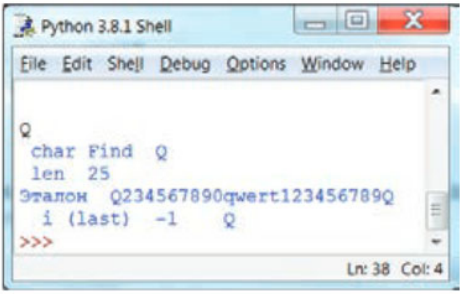
– дублирование строки (*);

– доступ по индексу ([i1,i2,i3]).

Покажем, как находить символ в строке. Приведем вариант, в котором реализован поиск без использования функции, но использующий оператор цикла *for*. Первая программа в строке (названа «Эталон») ищет первое вхождение заданного символа. Искомый символ задается с клавиатуры. Приводятся два результата. В первом скриншоте искомый символ находится на первом (левом месте). Но поиск начинается с номера (–1). Во втором скриншоте показано, как будет работать программа, если в строке два искомых символа. Они расположены на первом и последнем местах.

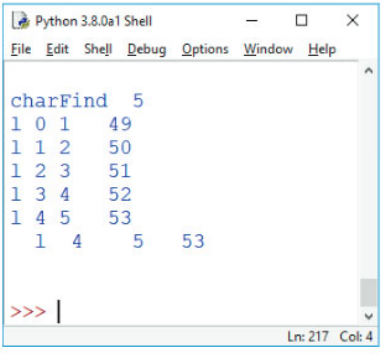
```
# charFind='Q'
charFind=input()
print(' char Find ', charFind)
Эталон='Q234567890qwert1234567890'
print(' len ', len(Эталон) )
i=-1
while( (Эталон[i] != charFind) and (i<len(Эталон)-1 ) ):
  i=i+1
  print(i, '  ', Эталон[i])
print('  i (last) ', i, '  ', Эталон[i])
```

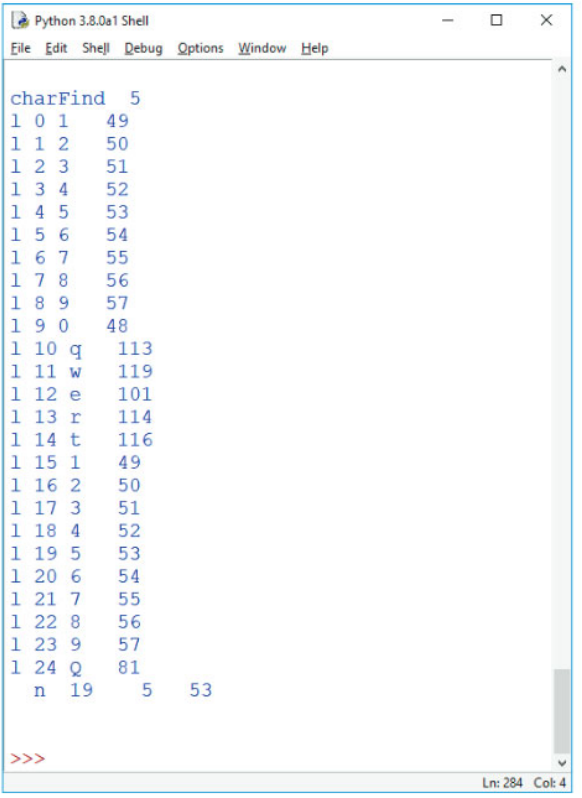




В следующем фрагменте программы искомый символ задается в программе, а не с клавиатуры. Условием нахождения символа является совпадение его кода с заданным целым числом.

```
print('\n')
Эталон='1234567890qwert123456789Q'
l=0; n=-1; charFind='5'
print('charFind ', charFind)
for i in Эталон:
  print('l',l, Эталон[l], ' ', ord(Эталон[l]) )
  if ( ord(Эталон[l]) == 53 ):
    n=l
    break
  l=l+1
if  n>-1:
  print(' l ', l, '  ', Эталон[l], ' ', ord(Эталон[l]))
else:
  print('NO')
print('\n')
```





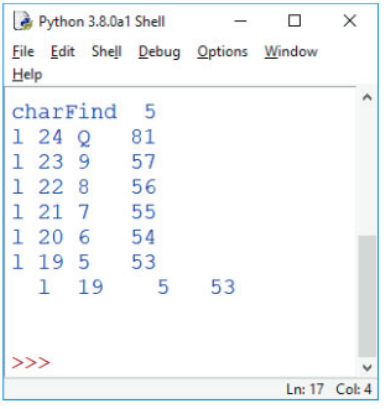
Теперь изменим программу так, чтобы она искала последнее вхождение заданного символа. Сначала сделаем это, изменив последний текст так: номер символа сохраняется в переменной l.

```
# Последнее вхождение символа
print('\n')
Эталон='1234567890qwert123456789Q'
l=0; n=-1; charFind='5'
print('charFind ', charFind)
for i in Эталон:
  print('l',l, Эталон[l], ' ', ord(Эталон[l]) )
  if ( ord(Эталон[l]) == 53 ):
    n=l
#     break
```

```
    l=l+1
  if  n>-1:
    print(‘  n ‘, n, ‘  ‘, Эталон[n], ‘ ‘, ord(Эталон[n]))
  else:
    print(‘NO’)
  print(‘\n’)
```

Сделаем программу более эффективной, начиная поиск с последнего элемента.

```
  # Последнее вхождение символа
  print(‘\n’)
  Эталон=’1234567890qwert123456789Q’
  n=-1; charFind=’5’
  print(‘charFind ‘, charFind)
  Длина=len(Эталон); l=Длина-1
  for i in range(Длина):
    print(‘l’,l, Эталон[l], ‘ ‘, ord(Эталон[l]) )
    if ( (ord(Эталон[l])) == 53 ):
      n=l
      break
    l=l-1
  if  n>-1:
    print(‘  l ‘, l, ‘  ‘, Эталон[l], ‘ ‘, ord(Эталон[l]))
  else:
    print(‘NO’)
  print(‘\n’)
```



6.2. Функции работы со строками

В Питоне есть много функций работы с символьными данными. Приведем примеры использования некоторых из них.

Сначала приведем фрагмент, использующий функции поиска: *find* и *rfind*. Они возвращают индекс (номер) найденной подстроки в исходной строке (если они есть) или (–1) в противном случае. Имя исходной строки задается в начале обращения к функции, после него ставится точка, а далее аргументы: подстрока для поиска, начальный индекс поиска. Вот пример синтаксиса функции.

find().str.find(str, beg = 0 end = len(string))

Вторая функция вызывается аналогично. У этих функций есть отличие в возвращаемом значении. Оно состоит в следующем. Про функцию *find*() говорится, что она определяет, входит ли подстрока в строку. Если строка найдется, то результат – номер первого вхождения. А вот *rfind*() – «возвращает последний индекс, в котором находится подстрока».

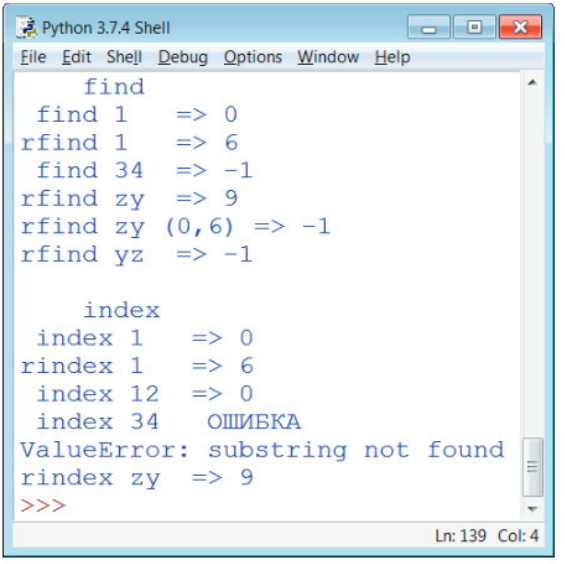
В приведенном примере исходная строка содержит цифровые и буквенные символы.

```
Строка = '123abc123zyx'
print(Строка)
print('  find')
print(' find 1  =>', Строка.find('1'))
print('rfind 1  =>', Строка.rfind('1'))
print(' find 34 =>', Строка.find('34'))
print('rfind zy =>', Строка.rfind('zy'))
print('rfind zy (0,6) =>', Строка.rfind('zy',0,6))
print('rfind yz =>', Строка.rfind('yz'))
```

Есть еще две функции (*index* и *rindex*), которые выполняют похожие действия. Но при отсутствии шаблона в исходной строке генерируется ошибка.

```
print('\n index')
print(' index 1  =>', Строка.index('1'))
print('rindex 1  =>', Строка.rindex('1'))
print(' index 12 =>', Строка.index('12'))
print(' index 34  ОШИБКА')
print('ValueError: substring not found')
```

```
print('rindex zy =>', Строка.rindex('zy')) ('zy'))
# index работает плохо
# print('rindex 34 in ', Строка.rindex('34'))
# ValueError: substring not found
```



Следующая группа: функции работы со строками выполняют различные проверки символьных строк. В исходном тексте фрагмента приведены такие функции:

isdigit() – состоит ли строка из цифр;

isalpha() – состоит ли строка из букв;

isalnum() – состоит ли строка из цифр или букв;

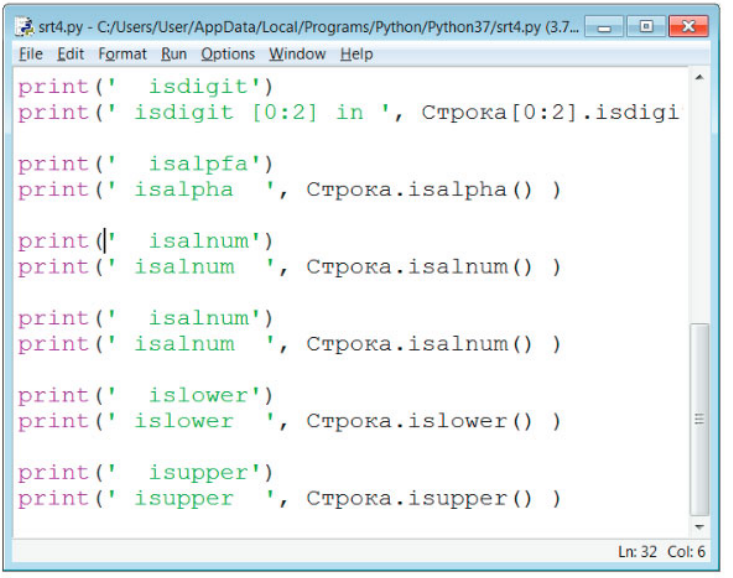
islower() – состоит ли строка из символов в нижнем регистре;

isupper() – состоит ли строка из символов в верхнем регистре.

Они обрабатывают ту же символьную строку, что и ранее.

```
print(,  isdigit')
print(, isdigit in ,, Строка.isdigit() )
print(,  isdigit')
print(, isdigit [0:2] in ,, Строка[0:2].isdigit() )
print('  isalpfa')
print(' isalpha  ', Строка.isalpha() )
```

```
print(‘  isalnum’)
print(‘ isalnum  ‘, Строка.isalnum() )
print(‘  isalnum’)
print(‘ isalnum  ‘, Строка.isalnum() )
print(‘  islower’)
print(‘ islower  ‘, Строка.islower() )
print(‘  isupper’)
print(‘ isupper  ‘, Строка.isupper() )
```

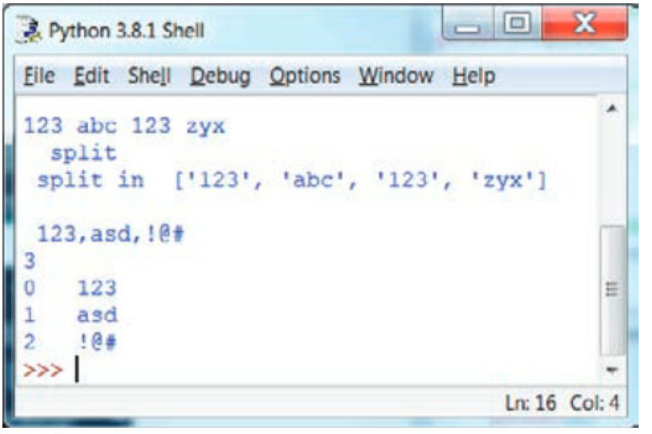


В этой же группе функций Питона по работе со строками есть еще и такие, которые определяют, состоит ли строка из символов верхнего или нижнего регистров, является ли первая буква строки заглавной и т.д.

Следующая функция для работы со строками Питона выделяет из строки ее регулярные части.

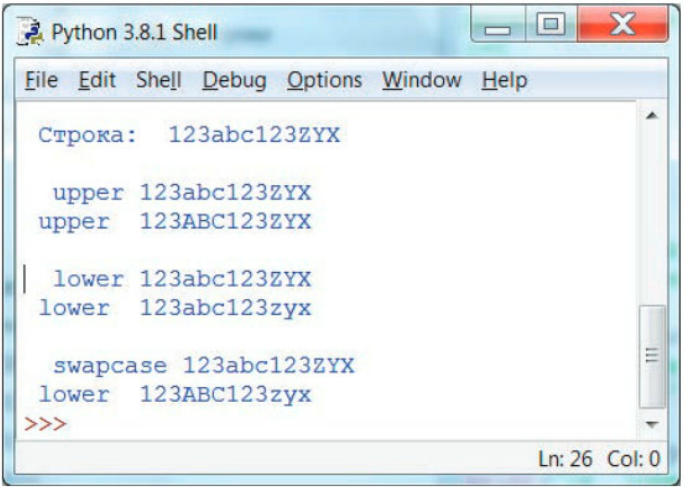
```
Строка = ‘123 abc 123 zyx’
print(Строка)
print(‘  split’)
print(‘ split in ‘, Строка.split(‘ ‘) )
subSet1=[‘ ‘]
```

```
stroka='123,asd,!@#'
print('\n',stroka)
subSet1=stroka.split(',')
print(len(subSet1))
i=0
while ( i<3 ):
  print(i, ' ', subSet1[i])
  i=i+1
# end while
```



Третья группа строковых: функция Питона преобразует строку символов, например в верхний или нижний регистр, а еще меняет последний. Примеры их использования приведены далее.

```
Строка = '123abc123ZYX'
print('\n Строка: ', Строка)
СтрокаUp = Строка.upper()
print('\n  upper', Строка)
print(' upper ', СтрокаUp )
СтрокаLow = Строка.lower()
print('\n  lower', Строка)
print(' lower ', СтрокаLow )
СтрокаUpLow = Строка.swapcase()
print('\n  swapcase', Строка)
print(' lower ', СтрокаUpLow)
```



Здесь рассмотрены не все функции работы со строками в Питоне. Теперь переходим к рассмотрению других типов данных Питона, с именем которых связано не одно значение, а несколько.

Экзаменационные темы

1. Строковые данные в Питоне. Равноправие кавычек.
2. Строковые данные в Питоне. Доступ к элементам.
3. Коды символов. Функции *ord* и *chr* Питона.
4. Методы работы со строками в Питоне.

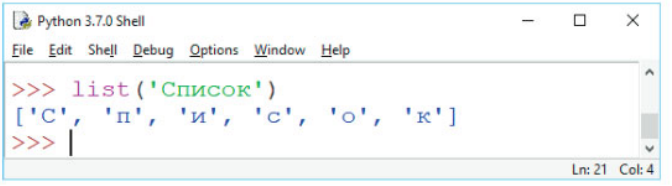
7. СПИСКИ – АНАЛОГ МАССИВОВ В ПИТОНЕ

В Питоне нет множественных данных, называемых массивами, как в большинстве языков программирования. Что такое «массив» в традиционном смысле? Это множество значений одного типа. Доступ к отдельному элементу осуществляется указанием имени, после которого надо фиксировать его положение (номер). Последний чаще начинается с нуля. Тогда в памяти его положение определяется смещением относительно начального адреса с учетом выделяемого участка памяти для элемента определенного типа. Так достигается эффективность обработки массивов данных. В Питоне этого нет. Списки являются множественными значениями. Каждый элемент списка может быть, как простым (с одним значением), так и множественным (со многими значениями).

Приведем пример нескольких списков. На сайте по адресу https://pythonworld.ru/tipy-dannyx-v-python/spiski-list-funkcii-i-metody-spiskov.html приводится такая команда, выполняемая в режиме оболочки Питона:

list('Список')

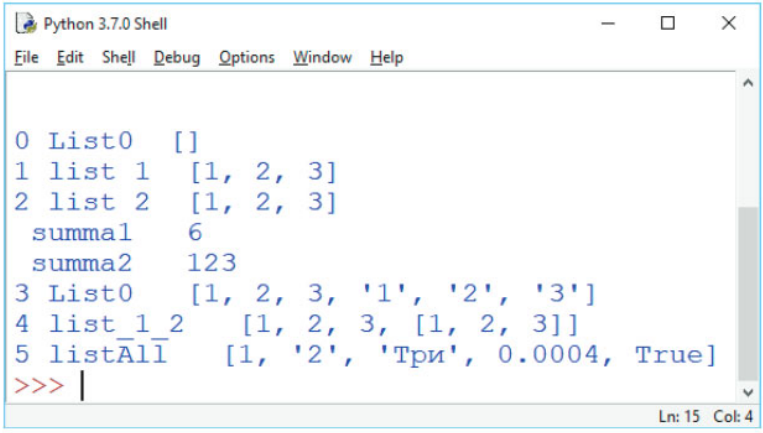
Тогда будет выведено следующее.



Теперь приведем текст, который сохраняется как файл Питона (его расширение *.py) и выполняется после нажатия клавиши F5. Создается три списка: пустой, с тремя целочисленными элементами и тремя символьными. Далее с ними выполняется операция сложения. Также показано, что элементами списка могут быть данные разных типов и другой список, а еще – как можно обращаться к его элементам.

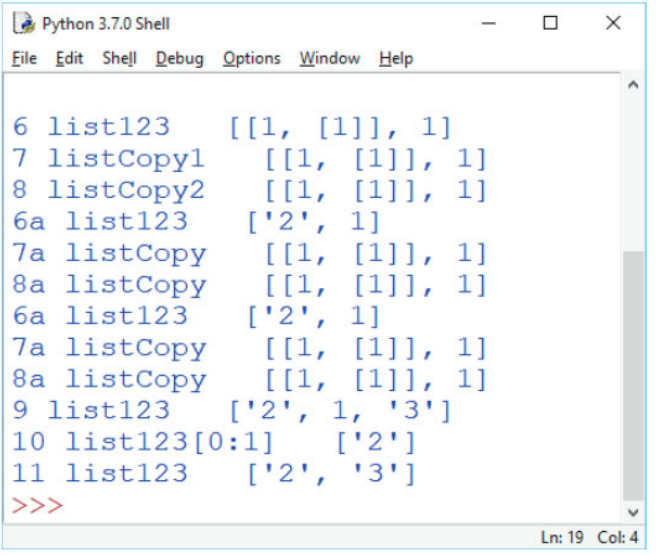
```
List0=[]
print('0 List0 ',List0)
```

```
list1=[1,2,3]
print('1 list 1 ',list1)

list2=["1",'2',"3"]
print('2 list 2 ',list1)

Sum1=list1[0]+list1[1]+list1[2]
print(' summa1  ',Sum1)

Sum2=list2[0]+list2[1]+list2[2]
print(' summa2  ',Sum2)

List0 = list1+list2
print('3 List0  ',List0)

list_1_2=[1,2,3,list1]
print('4 list_1_2  ',list_1_2)

ListAll=[1,'2',"Три",4e-4,True]
print('5 listAll  ',ListAll)
```




```
list123=[[1,[1]],1]
print('6 list123  ',list123)

listCopy1=list(list123)
print('7 listCopy1  ',listCopy1)
# listCopy2=list123
listCopy2=list123[:]
```

```
print('8 listCopy2 ',listCopy2)

list123[0]="2"
print('6a list123 ',list123)
print('7a listCopy ',listCopy1)
print('8a listCopy ',listCopy2)

list123[0]="2"
print('6a list123 ',list123)
print('7a listCopy ',listCopy1)
print('8a listCopy ',listCopy2)

list123.append("3")
print('9 list123 ',list123)

# Странно !!!
print('10 list123[0:1] ',list123[0:1])

del list123[1]
print('11 list123 ',list123)
```



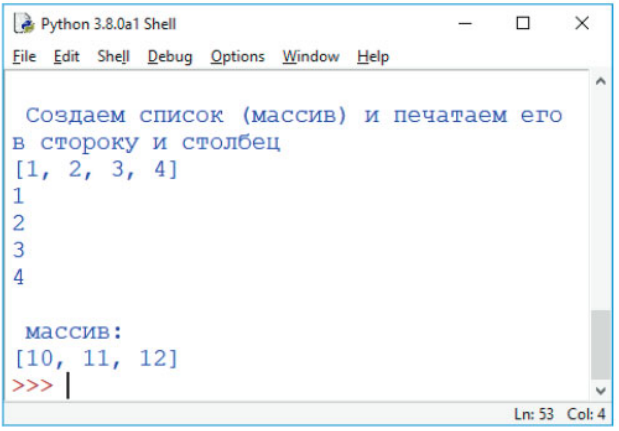
На интернет-ресурсе ***pythonworld*** приводится таблица, содержащая 10 методов работы со списками. Последние являются из-

меняемыми объектами. Поэтому для сохранения результата работы функции не требуется новой переменной.

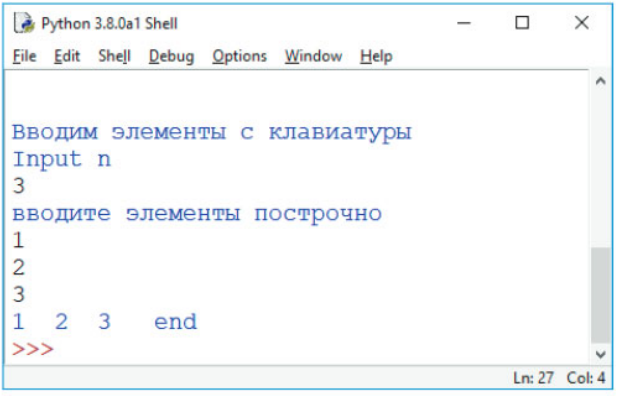
Метод	Что делает
list.append(x)	Добавляет элемент в конец списка
list.extend(L)	Расширяет список *list*, добавляя в конец все элементы списка *L*
list.insert(i, x)	Вставляет на *i*-й элемент значение *x*
list.remove(x)	Удаляет первый элемент в списке, имеющий значение *x*. *ValueError*, если такого элемента не существует
list.pop([i])	Удаляет *i*-й элемент и возвращает его. Если индекс не указан, удаляется последний элемент
list.index(x, [start [, end]])	Возвращает положение первого элемента со значением *x* (при этом поиск ведется от *start* до *end*)
list.count(x)	Возвращает количество элементов со значением *x*
list.sort ([key=функция])	Сортирует список на основе функции
list.reverse()	Разворачивает список
list.copy()	Поверхностная копия списка
list.clear()	Очищает список

Ранее говорилось, что в языковых конструкциях Питона не используется понятие массив. Задачи, реализующие типовые алгоритмы обработки таких данных, можно решать, используя списки. Для начала приведем несколько примеров, показывающих, как вводить элементы одномерного массива (списка) с клавиатуры.

```
print(‘ Создаем список (массив) и печатаем его’).
print(‘в стороку и столбец’)
List=[1,2,3,4]
print(List)
for i in range(len(List)):
   print(List[i])
print

# Списковое выражение
массив = [x for x in (10,11,12)]
print(‘\n массив: ‘)
print(массив)
```

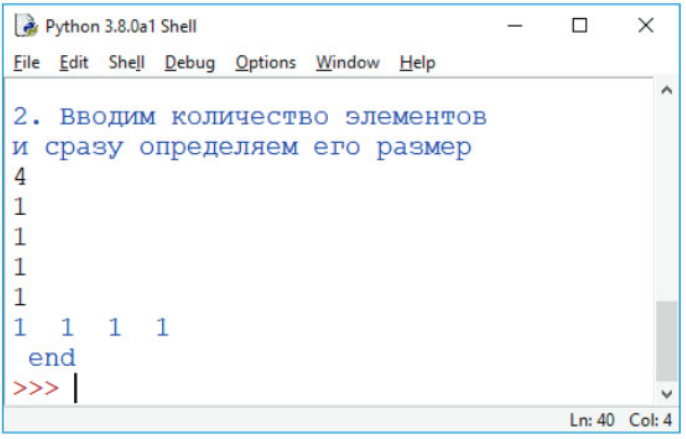


Есть относительно много способов генерировать или вводить с клавиатуры элементы массива. Приведем несколько примеров. Их описания даны в тексте программ как комментарии. В приведенных примерах используется сочетание символов end='', которое определяет то, что следующий оператор вывода (*print*) не начнется с новой строки.



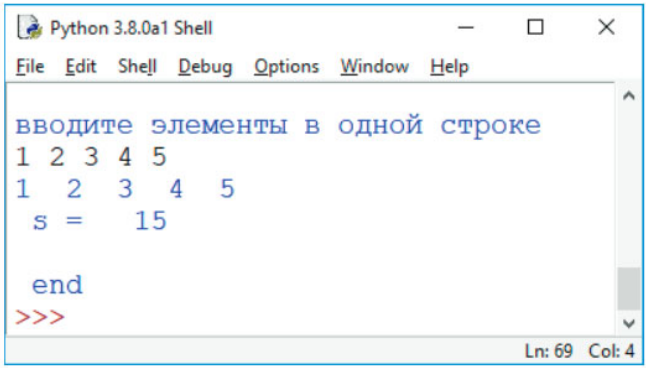

```
# Пример 1. Вводим количество элементов и далее значения.
print('Вводим элементы с клавиатуры')
print('Input n')
n=int(input())
```

```
a1 = [0] *n
print('вводите элементы построчно')
for i in range(len(a1)):
    a1[i] = int(input())
for i in range(len(a1)):
    print(a1[i], ' ', end='')
print(' end')
# Пример 2. Вводим количество элементов и сразу определяем его размер.
print('2. Вводим количество элементов \nи сразу определяем его размер')
a2 = [0] * int(input())
for i in range(len(a2)):
    a2[i] = int(input())
for i in range(len(a2)):
    print(a2[i], ' ', end='')
print('\n end')
```



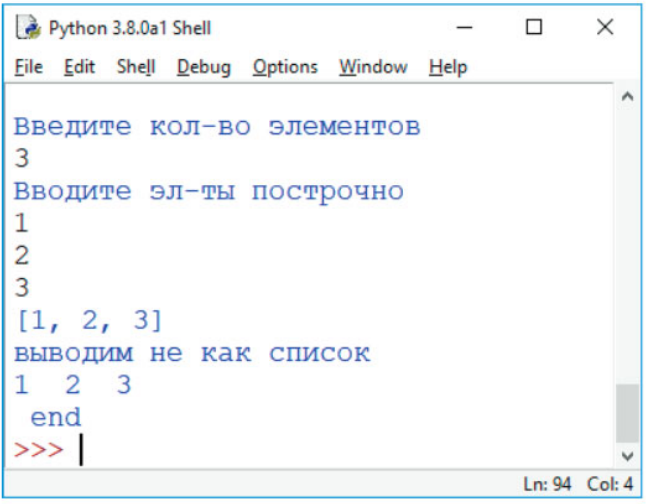

```
# Пример 3. Вводим элементы в одной строке (split разделяет символьную строку на элементы, разделенные пробелами).
print('вводите элементы в одной строке')
a3 = [int(s) for s in input().split()]
for i in range(len(a3)):
    print(a3[i], ' ', end='')
```

```
print()
s=0
for i in range(len(a3)):
  s=s+a3[i]
print(' s =  ',s)
print('\n end')
```

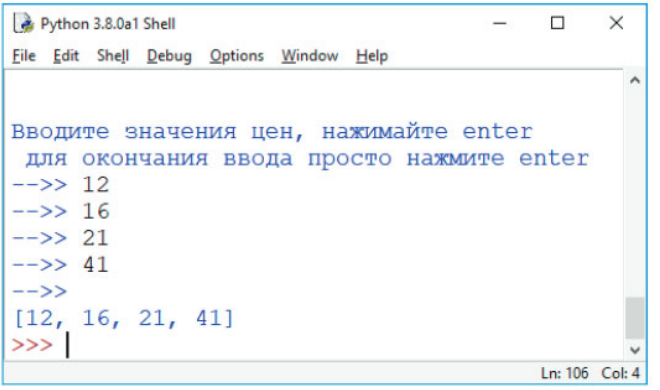


Пример 4. Сначала определим пустой список, затем пополняем (добавление – *append*).

```
a4 = []  # заводим пустой список
print('Введите кол-во элементов')
n = int(input())  # считываем количество элемент в списке
print('Вводите эл-ты построчно')
for i in range(n):
  new_element = int(input())   # считываем очередной элемент
  a4.append(new_element)      # добавляем его в список
                              # последние две строки можно было заменить одной:
                              # a.append(int(input()))
print(a4)
print('выводим не как список')
for i in range(len(a4)):
  print(a4[i], ' ', end='')
print('\n end')
```

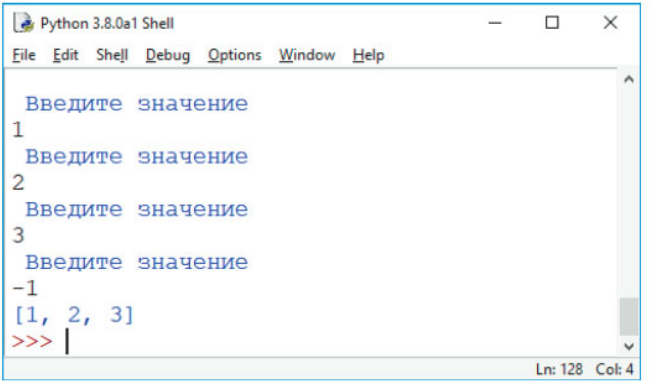



```
# Пример 5. Ввод элементов до нажатия enter (добавление – append).
# http://www.cyberforum.ru/python/thread2023600.html
print('Вводите значения и нажимайте enter')
print(' для окончания ввода просто нажмите enter')
a = int(input('-->> '))
rices = []
while True:
  try:
    rices.append(a)
    a = int(input('-->> '))
  except:
    break
print(rices)
```

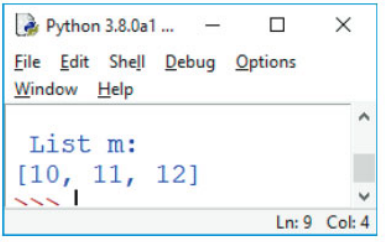


Пример 6. Ввод произвольного количества положительных чисел до –1 (для добавления используется *insert*).

```
addList2 = []
Ind  = 1; i=0
while ( Ind > 0):
  print(‘ Введите значение ‘)
  temp = int(input())
  if (temp>0):
    addList2.insert(i,temp)
    i=i+1
  else:
    Ind = -1
print(addList2)
```



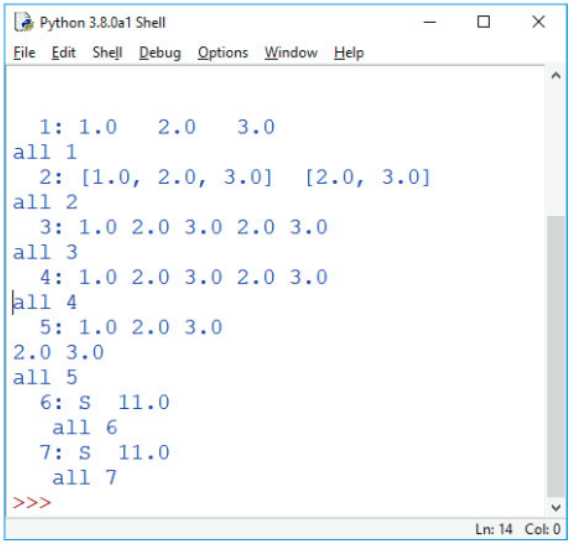
```
# Пример 7. Генерация массива с использованием спискового выражения.
массив = [x for x in (10,11,12)]
print(‘ List m: ‘)
print(массив)
```



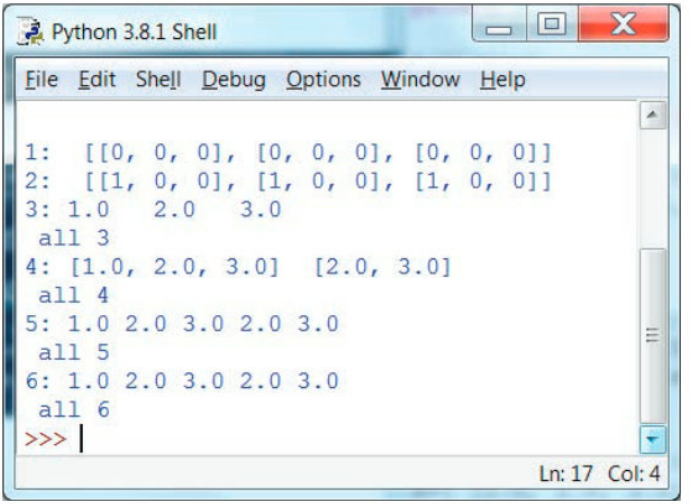
Теперь приведем несколько примеров работы со списками, являющимися аналогами двумерного массива.

```
# Двумерные массивы в Питоне:
print(‘  1: ‘,end=’’)
for a in [1.0, 2.0, 3.0]:
  print(a, ‘ ‘, end=’’)
print(‘\nall 1’)
print(‘  2: ‘,end=’’)
for a in [[1.0, 2.0, 3.0], [2.0,3.0]]:
  print(a, ‘ ‘, end=’’)
print(‘\nall 2’)
print(‘  3: ‘,end=’’)
A=[ [1.0, 2.0, 3.0], [2.0,3.0] ]
for i in range(len(A)):
  for j in range(len(A[i])):
    print(A[i][j], end=’ ‘)
print(‘\nall 3’)
print(‘  4: ‘,end=’’)
# То же самое, но циклы не по индексу, а по значениям списка:
for row in A:
  for elem in row:
    print(elem, end=’ ‘)
print(‘\nall 4’)
```

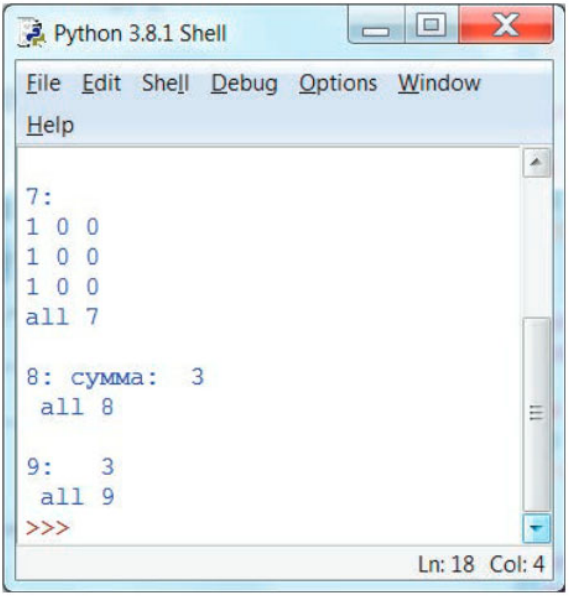
```
print(‘  5: ‘,end=’’)
# Естественно, для вывода одной строки можно воспользоваться методом join:
for row in A:
  print(‘ ‘.join(list(map(str, row))))
print(‘all 5’)
# Используем два вложенных цикла для подсчета суммы всех чисел в списке:
S = 0
for i in range(len(A)):
  for j in range(len(A[i])):
    S += A[i][j]
print(‘  6: S ‘, S )
print(‘   all 6’)
# Или то же самое с циклом не по индексу, а по значениям строк:
S = 0
for row in A:
  for elem in row:
    S += elem
print(‘  7: S ‘,S)
print(‘   all 7’)
```



```
    # Двумерные массивы в Питоне. Создаем массив 3 на 3.
    m=3; n=3
    A = [[0] * m] * n
    print('1 ', A)
    A[0][0]=1
    print('2 ', A)
    # Присвоили 1 A[0][0], а получили A[0][0] и A[1][0] A[2][0]
    print('  3: ',end='')
    for a in [1.0, 2.0, 3.0]:
      print(a, '  ', end='')
    print('\nall 3')
    print('  4: ',end='')
    for a in [[1.0, 2.0, 3.0], [2.0,3.0]]:
      print(a, ' ', end='')
    print('\nall 4')
    print('  5: ',end='')
    A=[ [1.0, 2.0, 3.0], [2.0,3.0] ]
    for i in range(len(A)):
      for j in range(len(A[i])):
        print(A[i][j], end=' ')
    print('\nall 5')
    print('  6: ',end='')
    # То же самое, но циклы не по индексу, а по значениям спи-
ска:
    for row in A:
      for elem in row:
        print(elem, end=' ')
    print('\nall 6')
```



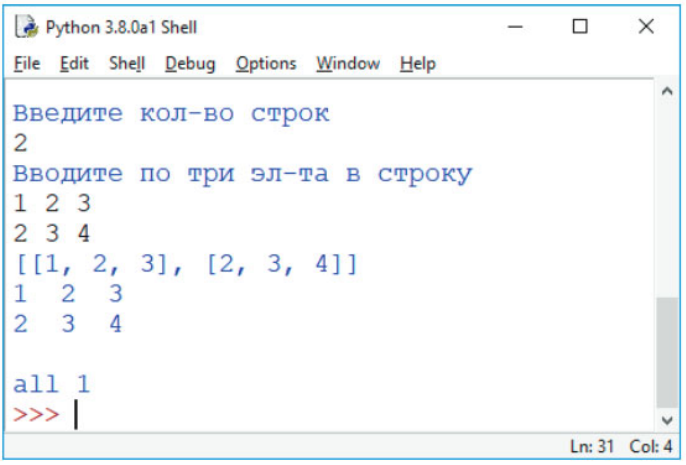

```
print(' 7: ',end='')
# Естественно, для вывода одной строки можно воспользоваться методом join:
for row in A:
  print(' '.join(list(map(str, row))))
print('all 7\n')
# Используем два вложенных цикла для подсчета суммы всех чисел в списке:
S = 0
for i in range(len(A)):
  for j in range(len(A[i])):
    S += A[i][j]
print(' 8 сумма: \n', S )
print(' all 8')
# Или то же самое с циклом не по индексу, а по значениям строк:
S = 0
for row in A:
  for elem in row:
    S += elem
print(' 9: сумма: ',S)
print(' all 9')
```



Покажем, как вводить элементы двумерного массива с клавиатуры. Для этого используется функция *split*, разделяющая элементы символьной строки, разделенные пробелами. Для добавления конкретного значения используется генератор. Следующие три примера сформированы после знакомства с материалом сайта https://pythontutor.ru/lessons/2d_arrays/.

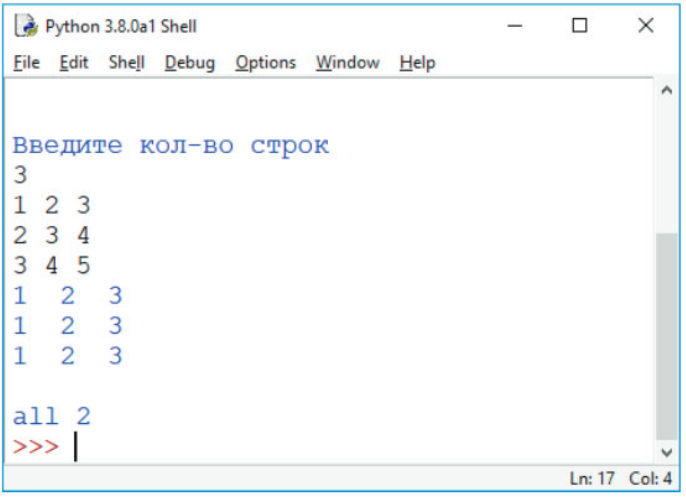
```
# Массив вводится построчно, например:
# 1 2 3
# 2 3 4
m=3
print('Введите кол-во строк')
n = int(input())
Array2 = []
print('Вводите по три эл-та в строку')
for i in range(n):
  Array2.append([int(j) for j in input().split()])
print(Array2)
for i in range(n):
  for j in range(m):
```

```
        print(Array2[i][j],' ',end='')
    print()
print('\nall ')
```



Во втором примере ввод организован без использования генератора.

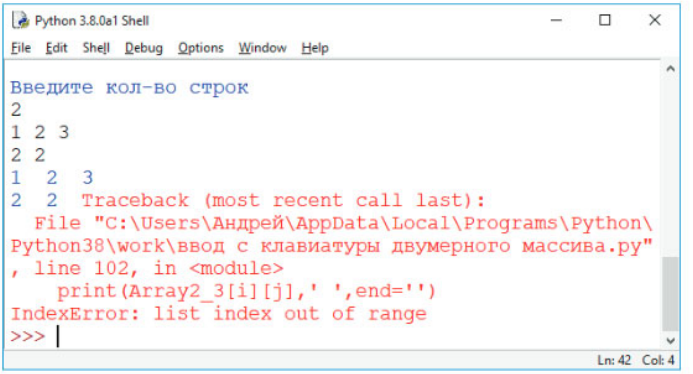
```
# 2)
m=3
print('Введите кол-во строк')
n = int(input())
Array2_2 = []
for i in range(n):
    row = input().split()
    for i in range(len(row)):
        row[i] = int(row[i])
        Array2_2.append(row)
#
for i in range(n):
    for j in range(m):
        print(Array2_2[i][j],' ',end='')
    print()
print('\nall 2')
```



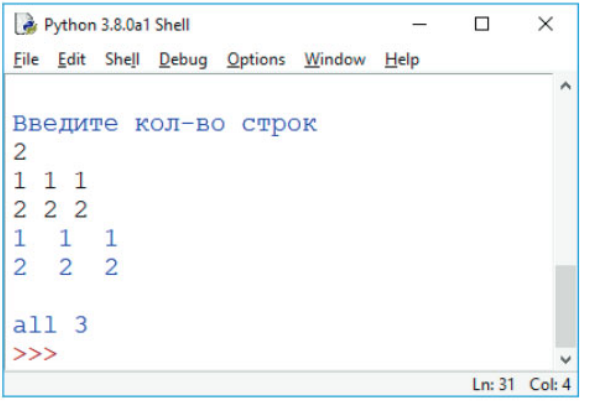
Можно сократить количество операторов, используя вложенные генераторы.

```
# 3)
m=3
print('Введите кол-во строк')
n = int(input())
Array2_3 = []
Array2_3 = [[int(j) for j in input().split()] for i in range(n)]
for i in range(n):
  for j in range(m):
    print(Array2_3[i][j],' ',end='')
  print()
print('\nall 2')
```

Ошибка при вводе элементов массива приведет к такому сообщению.



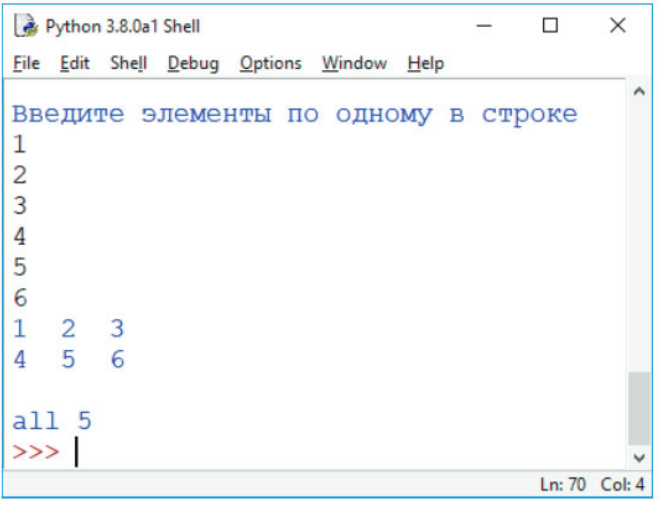
Теперь покажем, что будет выведено при правильном вводе массива.



В следующем примере элементы двумерного массива вводятся по одному в строке:

```
# 5)
n=2; m=3
Array4 = [[0 for j in range(m)] for i in range(n)]
print('Введите элементы по одному в строке')
for i in range(n):
  for j in range(m):
    Array4[i][j]=int(input())
for i in range(n):
```

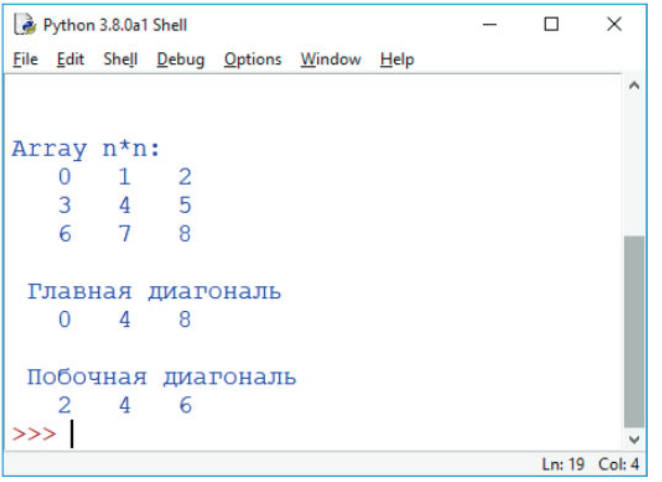
```
    for j in range(m):
      print(Array4[i][j],' ',end='')
    print()

print('\nall 5')
```



В следующем примере двумерный массив формируется как одномерный, а затем выводятся его главная и побочная диагонали:

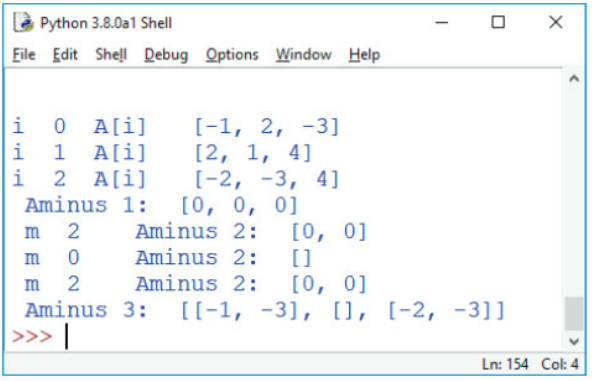
```
# Main diagonal
print("\n\nArray n*n:")
A1 = [0,1,2, 3,4,5, 6,7,8]
d=0; n=3
for i in range(n):
  for i in range(n):
    print(" %3d" %( A1[d]), end='' )
    d=d+1
  print();
d=0;
print(«\n Главная диагональ»);
for i in range(n):
  print(" %3d" %(A1[d]), end='')
  d=d+n+1
```

```
print()
# Revers diagonal
print("\n Побочная диагональ")
d=n-1
for i in range(n):
  print(" %3d" %(A1[d]), end='')
  d=d+n-1
print()
```



После того как мы познакомились с двумерными массивами, необходимо упомянуть о таком типе множественных данных, как ступенчатые массивы. Этот тип данных является обобщением таблицы (матрицы). Если в таблице каждая строка имеет одно и то же количество элементов, то у ступенчатого (двухмерного) массива в конкретной строке может быть определено свое количество элементов. В современных языках программирования такие типы данных, как правило, поддерживаются. Покажем, как это реализовано в Питоне. В рассмотренной далее задаче надо сформировать ступенчатый массив, выбирая в него только отрицательные значения исходной квадратной матрицы. Еще раз напомним, что как исходная матрица (или таблица), так и результирующая представляют собой список (*list*) в терминах Питона.

```
A = [ [-1,2,-3], [2,1,4], [-2,-3,4] ]
# print(' A: ', A)
for i in range(len(A)):
        print('i ',i, ' A[i]  ',A[i])
n=3
Aminus = [0] * n
print(' Aminus 1: ', Aminus)
for i in range(n):
   m=0
   for j in range(n):
      if A[i][j] < 0 :
         m=m+1
   Aminus[i] = [0] * m
print(' Aminus 2: ', Aminus)
# print(' Aminus first')
# print(Aminus)
for i in range(len(A)):
   l=0;
   for j in range(len(A[i])):
      if ( A[i][j] < 0):
#         print('i ',i, ' j ',j, ' l ',l)
        Aminus[i][l]=A[i][j]; l=l+1 # Yes-bad
print(' Aminus 3: ', Aminus)
```



Сделаем пояснения к тексту программы. После создания и вывода на экран исходного массива определено два цикла. В пер-

вом для каждой строки подсчитывается количество отрицательных элементов, а затем формируются заполненные нулями строки результирующего массива. Для второй строки (ее индекс 1) сформирован пустой список, потому что в исходном массиве в этой строке нет отрицательных элементов.

Экзаменационные темы

1. Два способа создания списков в Питоне.
2. Функции работы со списками.
3. Способы ввода значений одномерных массивов в Питоне.
4. Обработка списков как одномерных массивов.
5. Способы ввода значений двумерных массивов в Питоне.
6. Обработка списков как двумерных массивов.

8. КОЛЛЕКЦИИ – МНОЖЕСТВЕННЫЕ ТИПЫ ДАННЫХ

Что в принципе делает любой алгоритм? Изменяет значения переменных. К последним он обращается по имени. Но бывает так, что с именем связано не одно, а несколько значений. В Питоне данных такого типа может быть пять. В этом разделе будут рассмотрены основы создания и использования трех множественных типов данных: кортежей, словарей и множеств. Строки и списки были рассмотрены ранее.

Для данных Питона, с именем которых связано не одно, а несколько значений, в источниках используется несколько терминов: последовательность, коллекция, контейнер. Далее в нашем учебнике закрепим за таким типом данных термин «коллекция».

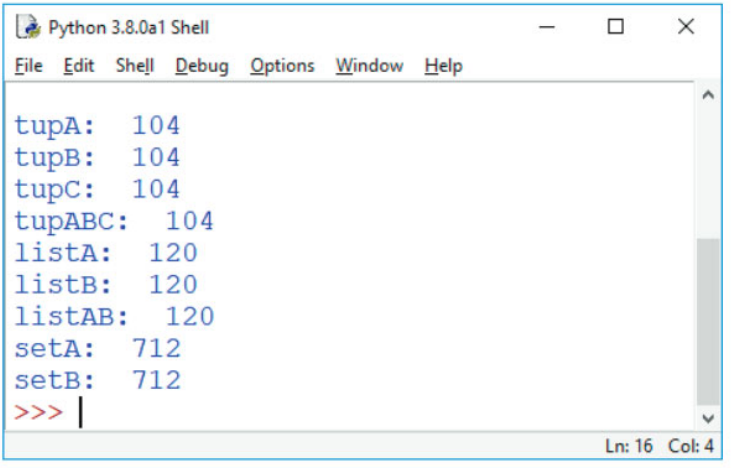
8.1. Кортежи

В этой части учебника подробнее рассмотрим три типа коллекций. Начнем с кортежей. Они представляют собой неизменяемые списки. Они похожи на константные списки. Кроме того, они занимают меньше места, чем изменяемые коллекции. Приведем фрагмент программы, в которой создаются кортежи, списки и множества с одинаковым количеством элементов (по десять).

```
print()
print()
tupA = ('0', '1', '2', '3', '4', '5', '6', '7', '8', '0')
tupB = (0,1,2,3,4,5,6,7,8,9)
tupC = (0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0)
tupABC = (0,1,2.0,3.0,'4.0',5.0,'6.0',7.0,8.0,9.0)
listA = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '0']
listB = [0,1,2,3,4,5,6,7,8,9]
listAB = [0,1,2,'3',4,5,'6',7,8,9.0]
setA = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '0'}
setB = {0,1,2,3,4,5,6,7,8,9}
print( 'tupA: ',tupA.__sizeof__() )
print( 'tupB: ',tupB.__sizeof__() )
```

```
print( 'tupC: ',tupC.__sizeof__() )
print( 'tupABC: ',tupABC.__sizeof__() )
print( 'listA: ',listA.__sizeof__() )
print( 'listB: ',listB.__sizeof__() )
print( 'listAB: ',listAB.__sizeof__() )
print( 'setA: ',setA.__sizeof__() )
print( 'setB: ',setB.__sizeof__() )
```

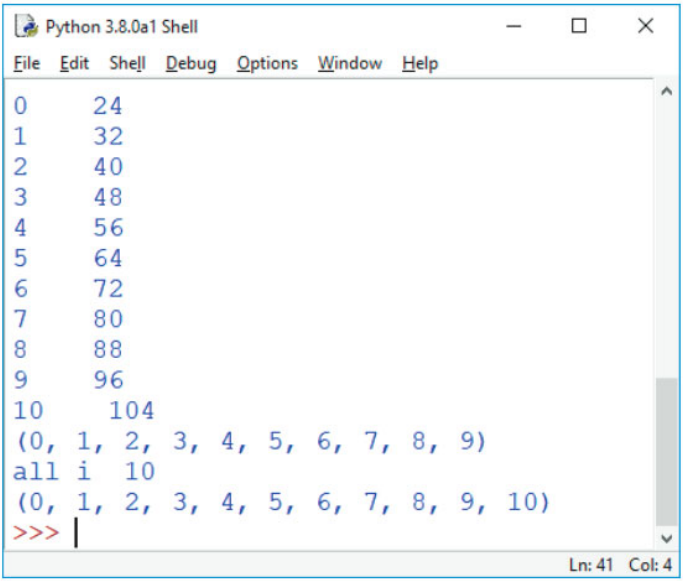
Такой фрагмент программы приведет к такому результату.



В примере определены четыре кортежа, три списка и два множества. Видно, что все три коллекции независимо от типа при одинаковом количестве элементов занимают одинаковое место в памяти. Для кортежей это 104 (наименьшее значение), для списка – 120, а для множества – 712. Заметим, что первый из кортежей содержит символьные элементы, второй – целые числа, третий – вещественные числа, а четвертый – данные разного типа. Так же и списки – состоят как из элементов одного типа, так и разного. Только множества составлены из элементов одного типа: первое – из символьных, а второе – из целых чисел. Смешивать типы элементов во множестве нельзя.

В следующем фрагменте программы сначала формируется начальный кортеж из одного элемента, далее к нему последовательно добавляется по одному элементу. Длина полученно-

го кортежа постоянно выводится. Видно, что длина начального кортежа 24, а каждый следующий больше предыдущего на 8.

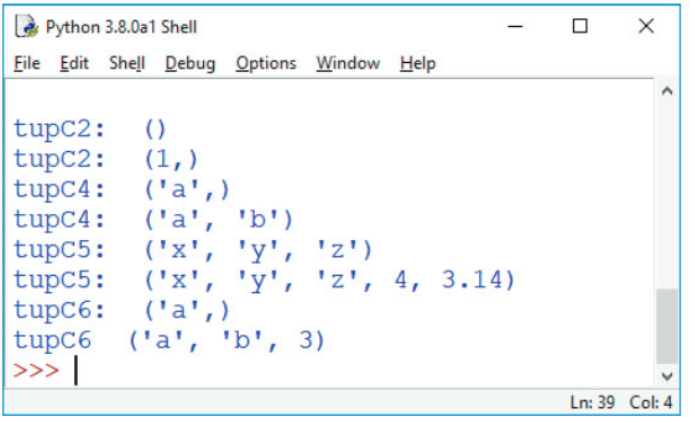


Ранее говорилось, что кортежи объявлены как неизменяемый тип данных. Это означает, что в созданном кортеже нельзя изменять значения уже сформированных элементов. Но к кортежу можно добавлять новые элементы (только в «хвост»). Обращаем внимание, что для этого надо новое значение заключить в круглые скобки и добавить запятую. Недопустимые способы добавлять элементы кортежа в следующем фрагменте оформлены как комментарии.

```
# tupC1 = tuple()
# NO tupC1 = tupC1 + (1)
# TypeError: can only concatenate tuple (not “int”) to tuple
tupC2 = ()
print( ‘tupC2: ‘,tupC2 )
tupC2 = tupC2 + (1,)
print( ‘tupC2: ‘,tupC2 )
# NO tupC3 = tuple(10)
# NO tupC3 = tupC3 + (11)
```

```
# TypeError: can only concatenate tuple (not “int”) to tuple
tupC4 = (“a”,)
print( ‘tupC4: ‘,tupC4 )
tupC4 = tupC4 + (“b”,)
print( ‘tupC4: ‘,tupC4 )
tupC5 = ‘x’,’y’,’z’
print( ‘tupC5: ‘,tupC5 )
tupC5 = tupC5 + (4,) + (3.14,)
print( ‘tupC5: ‘,tupC5 )
tupC6 = tuple(“a”)
print( ‘tupC6: ‘,tupC6 )
tupC6 = tupC6 + (“b”,) + (3,)
print( ‘tupC6: ‘,tupC6 )
```

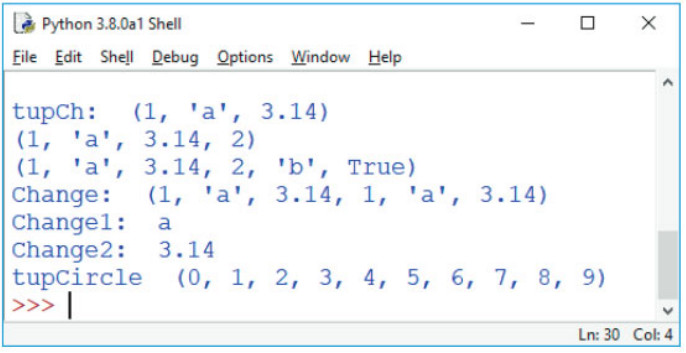
Во так выглядит результат.



Обращаем внимание, что элементы кортежа могут иметь разный тип (целое и символьное). Следующий пример развивает этот факт. Кроме того, в нем показано, что при формировании кортежа можно не включать набор его значений в круглые скобки. Также в заключении показано, как создавать кортеж, используя итератор.

```
tupCh=(1,’a’,3.14,)
print(‘tupCh: ‘, tupCh)
tupCh1=tupCh
```

```
tupCh1=tupCh1+(2,)
print(tupCh1)
tupCh1=tupCh1+('b',)+(True,)
print(tupCh1)
tupCh=tupCh*2
print('Change: ', tupCh)
print('Change1: ', tupCh[1])
print('Change2: ', tupCh[-1])
tupCircle=a = tuple(i for i in range(0, 10))
print('tupCircle ', tupCircle)
```



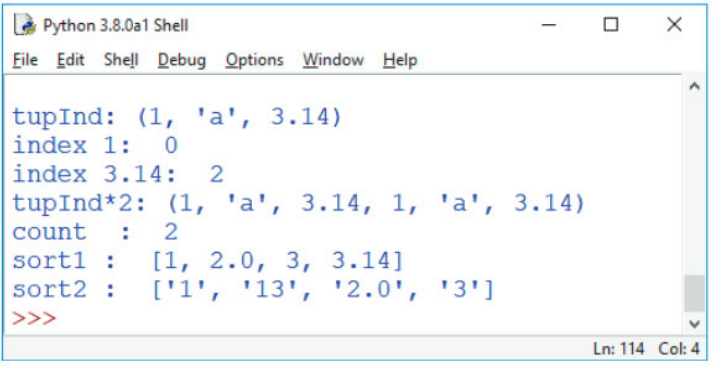
Теперь покажем, как можно обращаться к элементам кортежа по значению. Также из фрагмента видно, что обращение по номеру элемента и изменение значения недопустимо даже для вывода значения, и тем более для его изменения. Операция + уже рассматривалась ранее. А операция * дублирует элементы кортежа. В нашем примере –удваивает.

Сообщим, что для обработки кортежей есть два метода:
index () – возвращает индекс (индексы) в кортеже по заданному значению (если значения нет, выводится сообщение об ошибке);
count () – возвращает число, показывающее, сколько раз заданное значение встречается в кортеже.

В приведенном далее примере также показано, что к кортежу можно применить функцию сортировки. Но ее можно применять только для данных одного типа (числовых или символьных).

```
tupInd=(1, 'a', 3.14, )
```

```
print('tupInd:',tupInd)
print('index 1: ', tupInd.index(1) )
print('index 3.14: ', tupInd.index(3.14) )
# print('index 2: ', tupInd.index(2) ) -
# ValueError: tuple.index(x): x not in tuple
# tupInd[0]=2*tupInd[0]
# TypeError: 'tuple' object does not support item assignment
print('tupInd*2: ', end='')
tupInd=tupInd*2
print(tupInd)
print('count  : ',tupInd.count(1) )
print('index 1: ', tupInd.index(3.14) )
# tupInd=sorted(tupInd)
#print('NO sort : ', tupInd)
# TypeError: '<' not supported between instances of 'str' and 'int'
tupSort1=( 3.14, 1, 3, 2.0)
tupSort1=sorted(tupSort1)
print('sort1 : ', tupSort1)
tupSort2=( '3', '1', '13', '2.0')
tupSort2=sorted(tupSort2)
print('sort2 : ', tupSort2)
```



Обращаем внимание на то, что допустима сортировка целых и вещественных чисел в одном кортеже, а также на особенность сортировки символьных элементов кортежа (в строке sort2).

8.2. Словари

Теперь рассмотрим подробнее, как определять и использовать словари. Словари определяются как неупорядоченная последовательность. Доступ к элементам последовательности осуществляется по ключу. Есть четыре способа создания словарей:

– используя литералы;

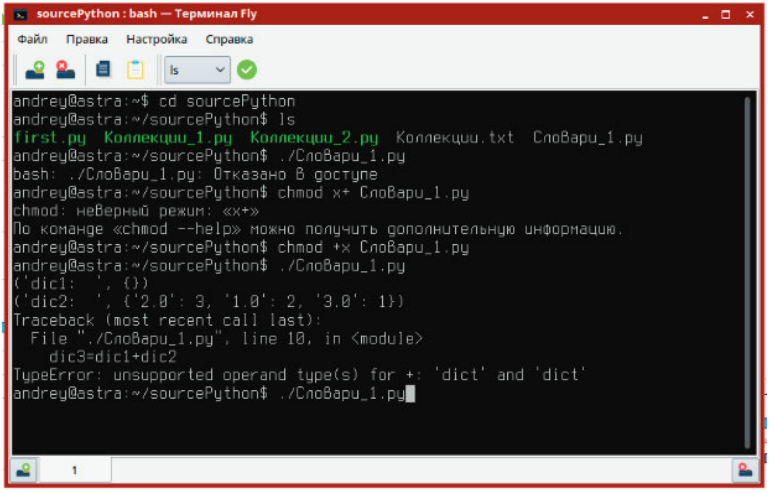
– применяя ключевое слово *dict*;

– применяя генератор;

– используя *dict.fromkey* (неизменяемые словари).

Приведенные примеры демонстрируют эти способы создания словарей. Они выполнены в режиме *Live CD* операционной системы *Alt Linux* Образование версии 8.

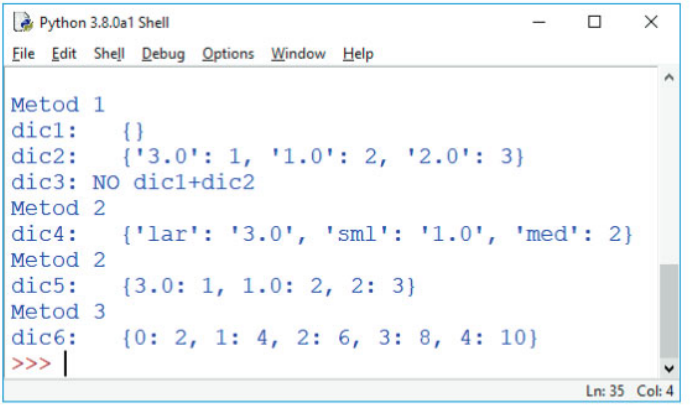
```
#!/usr/bin/python
print(‚Metod 1‘)
dic1={}
print(‘dic1: ‘, dic1)
dic2={‘3.0’:1, ‘1.0’:2, ‘2.0’:3 }
# NO dic2={lar=’3.0’, sml=’1.0’, med=2}
print(‘dic2: ‘, dic2)
# dic3=dic1+dic2
# TypeError: unsupported operand type(s) for +: ‘dict’ and ‘dict’
# print(‘dic3: ‘, dic2)
```



Описанные примеры со словарями создавались вне среды разработки. Исходный текст фрагмента программы формировался в текстовом редакторе. Это может быть стандартная утилита *vi* командного режима или редактор *Kate* режима графического интерфейса. В примере показано, что после создания кода программы надо придать ему свойства «исполнимый». Это выполняется командой *chmod*.

После запуска программы создаются два словаря: один не содержит элементов, а второй содержит три элемента. Затем показано, что операция «+» для словарей недопустима.

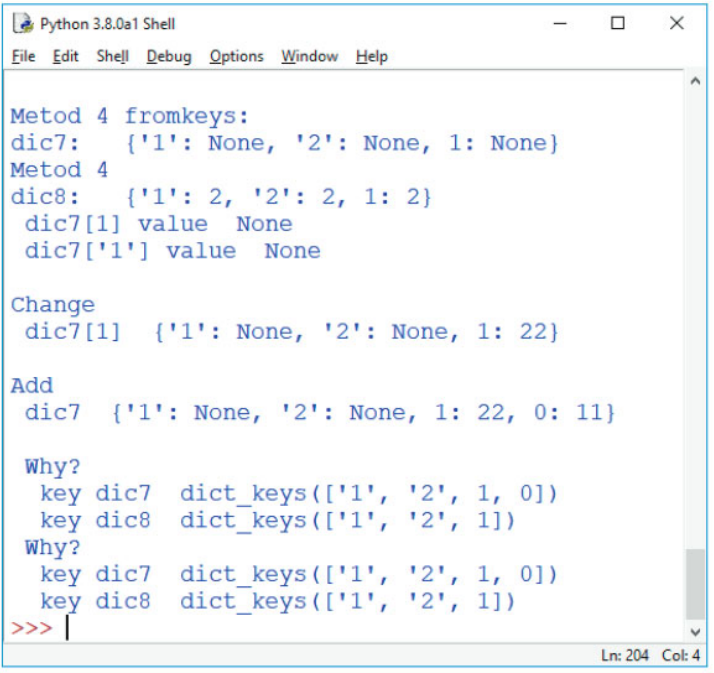
```
print('Metod 2')
dic4=dict(lar='3.0', sml='1.0', med=2)
print('dic4: ', dic4)
print('Metod 2')
dic5=dict([(3.0,1), (1.0,2), (2,3)])
print('dic5: ', dic5)
print('Metod 3')
dic6={j: (j+1)*2 for j in range(5) }
print('dic6: ', dic6)
```



Используя метод *dict.fromkeys*(), получим словарь с одинаковым значением ключа. Также значение словаря можно изменять. Это демонстрируется следующим примером.

```
print('1: ')
dic1={}
```

```
print('dic1:  ', dic1)
dic2={'3.0':1, '1.0':2, '2.0':3 }
# NO dic2={lar='3.0', sml='1.0', med=2}
print('dic2:  ', dic2)
# dic3=dic1+dic2
print('dic3: NO dic1+dic2')
# TypeError: unsupported operand type(s) for +: 'dict' and 'dict'
print('\n2: ')
dic4=dict(lar='3.0', sml='1.0', med=2)
print('dic4:  ', dic4)
print('\n3: ')
dic5=dict([(3.0,1), (1.0,2), (2,3)])
print('dic5:  ', dic5)
print('\n4: ')
dic6={j: (j+1)*2 for j in range(5) }
print('dic6:  ', dic6)
print('Metod 4 fromkeys:')
dic7=dict.fromkeys(['1', '2', 1, True])
print('dic7:  ', dic7)
print('\n5: ')
dic8=dict.fromkeys(['1', '2',  1, True], 2)
print('dic8:  ', dic8)
print(' dic7[1] value ', dic7[1])
print(" dic7['1'] value ", dic7['1'])
print('\nChange')
dic7[1] = 22
print(' dic7[1] ', dic7)
print('\nAdd')
dic7[0] = 11
print(' dic7 ', dic7)
print()
# https://devpractice.ru/python-lesson-9-dict/
print(' Why?')
print('  key dic7 ', dict.keys(dic7))
print('  key dic8 ', dict.keys(dic8))
print(' Why?')
print('  key dic7 ', dic7.keys())
print('  key dic8 ', dic8.keys())
```



В этом разделе учебника уже упоминался ресурс ***pytonworld***. На его странице приводится такой список методов словарей.

dict.clear() – очищает словарь.

dict.copy() – возвращает копию словаря.

classmethod **dict.fromkeys**(seq[, value]) – создает словарь с ключами из *seq* и значением *value* (по умолчанию *None*).

dict.get(key[, default]) – возвращает значение ключа, но если его нет, не бросает исключение, а возвращает *default* (по умолчанию *None*).

dict.items() – возвращает пары (ключ, значение).

dict.keys() – возвращает ключи в словаре.

dict.pop(key[, default]) – удаляет ключ и возвращает значение. Если ключа нет, возвращает *default* (по умолчанию бросает исключение).

dict.popitem() – удаляет и возвращает пару (ключ, значение). Если словарь пуст, бросает исключение *KeyError*. Помните, что словари неупорядочены.

dict.setdefault(key[, default]) – возвращает значение ключа, но если его нет, не бросает исключение, а создает ключ со значением *default* (по умолчанию *None*).

dict.update([other]) – обновляет словарь, добавляя пары (ключ, значение) из *other*. Существующие ключи перезаписываются. Возвращает *None* (не новый словарь!).

dict.values() – возвращает значения в словаре.

8.3. Множества

Множество – это коллекция данных, в которых нет повторяющихся элементов. Множества можно создать, используя ключевое слово *set* или фигурные скобки. Приведем еще два факта относительно множеств: они не поддерживают индексацию и элементы множества могут иметь разный тип. А еще для создания множеств можно использовать генератор. Покажем это на примерах.

```
s=set()
print(‘ set(): ‘,s)
print(‘ type() ‘, type(s) )
s={}
print(‘ (): ‘,s)
print(‘ type() ‘, type(s) )
# s=s.add(1) # Ошибка
# AttributeError: ‘dict’ object has no attribute ‘add’
s=set({1,2})
print(s)
# s=set(1)            # Ошибка
# TypeError: ‘int’ object is not iterable s=set(1)
#
# s=set(1,2)          # Ошибка
# s=set(True, True, False) # Ошибка более одного значения
# TypeError: set expected at most 1 argument, got 2
s=[True, True, False]
s=set(s) # но можно из списка
print(s)
# print(s[0])         # Ошибка
# TypeError: ‘set’ object is not subscriptable
```

```
s=set('Hello, world')      # Один элемент
print('s ', s)
s={'Hello,', 'world'}      # Два элемента
print('s ', s)
print(' len ', len(s))     # Вывод длины
print(' 2 in s3 ', 2 in s)  # Проверка наличия эл-та во множестве
s = {(j+1)* 2 for j in range(10)}  # Использование генератора
print('s  ', s)
```

Из приведенного скриншота становится понятно, что для создания пустого множества можно использовать метод *set*. А вот пустые фигурные скобки приводят к созданию пустого списка, к которому нельзя применить метод добавления элемента к множеству. В приведенном примере показано, как можно задать множество, расположив элементы в фигурных скобках (одной строки символов и двух строк). А еще можно элементы множества преобразовать из списка, а также вывести количество элементов множества и определить наличие в нем заданного значения. Демонстрируется применение генератора для формирования множества.

Это текст показывает следующие возможности работы со множествами: добавление и удаление элементов.

```
s={'1', 'a'}
print('s          ', s)
# add
s.add(1)              # Так можно добавить элемент во множество
print('s  add 1    ', s)
# s.add({2}) # А так добавлять элемент нельзя
# s=s+{2}    # И так добавлять элемент нельзя
# update
s.update('2')
print("s update '2'  ",s)
#  discard, remove
s.discard(1)
print('s discard 1   ',s)
s.discard(1)
print("s diccard '5' ",s)
s.remove('1')
print("s  remove '1' ",s)
```

```
# s.remove('5')  # Ошибка
# KeyError: '5'
# update
s.update({True,False}, {'x'}, {'y'})
print(«s update     «,s)
```

Но удалять элементы из множества можно еще и методами *pop* и *clear*. Конструкция *del* удаляет множество как объект, а не его элементы. Также показано, как можно получить доступ к элементам множества.

```
# pop
sp={1, 1.0, '1', None, False}
print('sp    ', sp, ' len ', len(sp) )
sp.pop()
print('sp  pop', sp, ' len ', len(sp) )
sp.pop()
print('sp  pop', sp, ' len ', len(sp) )
# clear
sc={'1', 3.14, 'Pi', '1', True}
print('(sc      ', sc)
sc.clear()
print('(sc clear) ', sc)
del sp
# print('sp  del', sp) #
# NameError: name 'sp' is not defined
print()
s={1, 1.0, '1', None, False}
print('s elements ',s)
for i in s:             #  Доступ к элементам множества
      print(i)
```

Теперь рассмотрим, как выполнять объединение множеств.

```
# unoin
s1={1,2,3}
print('s1  ', s1)
s2={3,4}
print('s2  ', s2)
s1.union(s1)          # Объединение множества самого с собой
print('s1 union s1', s1)
```

```
s2.union(s1)          # Объединение двух множеств
print('s2 union s1 ', s2)
s31={1}
s32={2}
s33={3}
s3= s31 | s32 | s33  #   Объединить множества можно и так
print('s3    ', s3)
```

Есть еще два метода, реализующие теоретико-множественные операции над множествами:

множество1.intersection(множество2) (пересечение – возвращает новое множество, содержащее все элементы, которые есть и в первом, и во втором множестве);

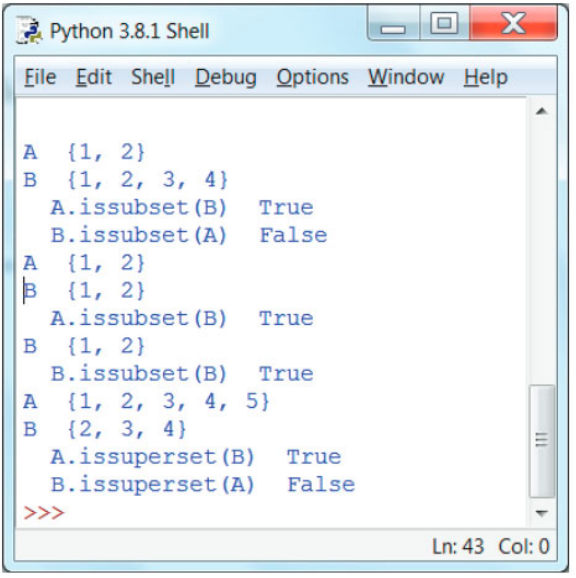
множество1.difference(множество2) (разность – возвращает новое множество, содержащее все элементы, которые есть в первом множестве, но которых нет во втором).

Приведем примеры их использования.

```
s1={1,2,3}
print('s1  ', s1)
s2={3,4}
print('s2  ', s2)
S12=s1.intersection(s2)
print('s1.intersection(s2)  ', S12)
S21=s2.intersection(s1)
print('s2.intersection(s1)  ', S21)
S11=s1.intersection(s1)
print('s1.intersection(s1)  ', S11)
print()
S12=s1.difference(s2)
print('s1.difference(s2)  ', S12)
S21=s2.difference(s1)
print('s2.difference(s1)  ', S21)
S22=s2.difference(s2)
print('s2.difference(s2)  ', S22)
```

В Питоне определены два метода, интерпретирующие два понятия: подмножество (*issubset*) и надмножество (*issuperset*). Приведем фрагмент программы, поясняющий возможности этих методов.

```
set1={1,2}
set2={2,3,4}
# A.issubset(B) A<=B  9
# True,  если A подмножество B
A=set1|set1
B=set2|set2|{4}|{1}
print('A ',A)
print('B ',B)
tf=A.issubset(B)
print('  A.issubset(B) ', tf)
tf=B.issubset(A)
print('  B.issubset(A) ', tf)
A=set1
B=set1
print('A ',A)
print('B ',B)
tf=A.issubset(B)
print('  A.issubset(B) ', tf)
B=set1
print('B ',B)
tf=B.issubset(B)
print('  B.issubset(B) ', tf)
# A.issuperset(B) A>=B  10
# True,  если A надмножество B
A=set1|set1|{1}|{3}|{5}|{4}
B=set2|set2
print('A ',A)
print('B ',B)
tf=A.issuperset(B)
print('  A.issuperset(B) ', tf)
tf=B.issuperset(A)
print('  B.issuperset(A) ', tf)
```



В приведенном фрагменте демонстрируются возможности применения отдельных символов-операторов, которые приводят к таким результатам, как и некоторые методы. Так, символ «|» реализует объединения, & – пересечение множеств, а <= фиксирует, является ли одно из множеств подмножеством другого и наоборот.

Закончим рассмотрение множественных данных в Питоне.

Ранее были приведены примеры использования функций в Питоне. Перейдем к рассмотрению вопроса написания собственных функций в этом языке программирования.

Экзаменационные темы

1. Создание кортежей в Питоне.
2. Операции над кортежами в Питоне.
3. Способы создание словарей в Питоне.
4. Методы работы со словарями в Питоне.
5. Создание множеств в Питоне.
6. Операции и методы множеств.

9. ФУНКЦИИ В ПИТОНЕ

Для чего создаются функции? Пожалуй, для того чтобы оформить в виде законченного фрагмента исходного текста программы какую-то часть общего алгоритма решения задачи. Но эта часть, как правило, должна обладать свойствами алгоритма: получать исходные данные и перерабатывать (преобразовывать) их в результат – выходные данные. При написании алгоритма исходный текст функции должен быть ограничен специальными метками. В языке Паскаль эту роль играют слова *Begin* и *End*. В ряде языков это фигурные скобки (открывающая и закрывающая). В языке Питон исходный текст, описывающий алгоритм, реализованный в функции, оформляется, как и у цикла или условной конструкции – отступами. Этим строкам исходного текста должна предшествовать строка описания, назовем ее заголовочной. Приведем пример. В этом фрагменте определена функция, которая получает число и возвращает его удвоенное значение. Ее исходный текст занимает первые две строки. Следующая строка показывает, как вызвать, исполнить функцию.

```
def func1(a):
    return 2*a
Rez=func1(2)
print(‘ 2*a ‘, Rez )
```

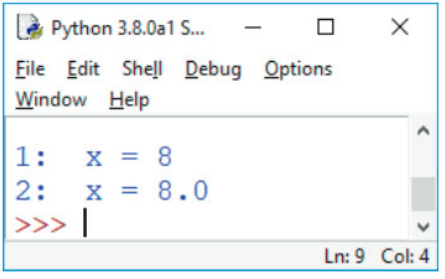
Рассмотрим структуру кода программы с функцией подробнее. В заголовочной строке функции должны присутствовать такие элементы: ключевое слово *def*, имя функции, круглые скобки, двоеточие. В круглых скобках можно ничего не записывать. Но в нашем примере записано имя входного аргумента – переменной. Далее записана только одна строка – тело функции. Там записан оператор, возвращающий удвоенное значение входной переменной.

В работе с функцией есть два этапа: определение и использование. Первый этап нашего примера описан. А вот использование функции записано в третьей строке. В ней переменной *Rez* присваивается значение, вычисляемое в *func*1. Очевидно, что при этом сначала выполняются операторы, составляющие тело функции, а затем – присваивание.

Если говорить о Питоне, то в нем можно определить два типа функции. Исходный текст именной функции, как было показано выше, начинается строкой с ключевым словом *def*, чего не предполагается для другого типа функций – анонимных. Их рассмотрим позже.

Продолжим рассмотрение примеров написания программ с именными функциями. Вернемся к первому примеру. Немного изменим его текст. Тело функции состоит из двух строк: вычисление значения переменной (оператор присваивания) и возврат (*return*) к вызвавшему ее коду. Далее эта функция выполняется дважды. Первый раз удваивается целочисленное значение, а второй – вещественное.

```
def f2(a):
  r=a*2
  return r
x=4
x=f2(x)
print('1: x =',x)
turn
x=4.0
x=f2(x)
print('2: x =',x)
```



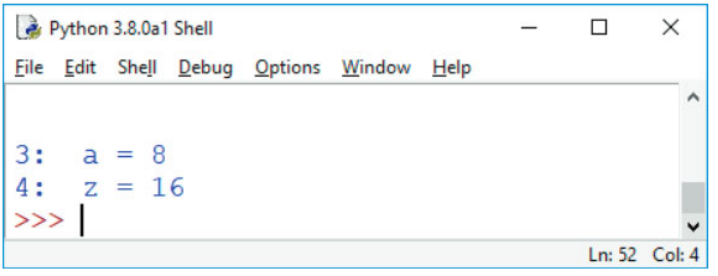
Данный пример показывает, что аргументы функции в Питоне не имеют типа. Это отличается от того, как это реализовано в других языках программирования, где тип аргументов функции строго определяется.

При написании программ с функциями исходные данные «фиксируются» дважды. В заголовке функции они представле-

ны формальным аргументом. Для него определен алгоритм преобразования исходных данных в результат. В нашем примере имя этой переменной *a*. При задании значений исходных данных вне тела функции «фиксируется» другая (или другие) переменная, называемая фактическим аргументом. В нашем примере таким аргументом является *x*.

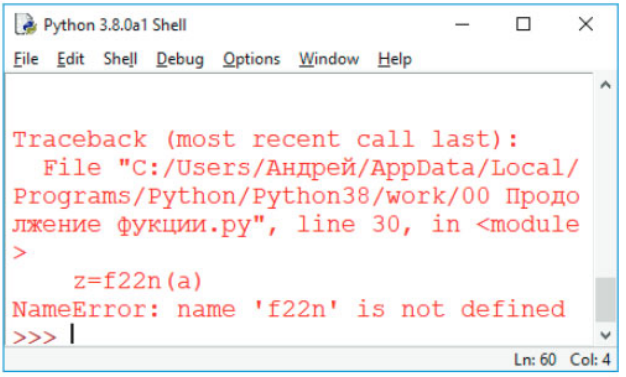
Следующий пример показывает, что имя формального и фактического аргумента может совпадать.

```
def f22(a):
  a=a*2
  return a
a=4
a=f22(a)
print('3: a =',a)
z=f22(a)
print('4: z =',z)
```



Если расположить новую функцию (ее исходный код) после операторов, вызывающих ее, получим ошибку.

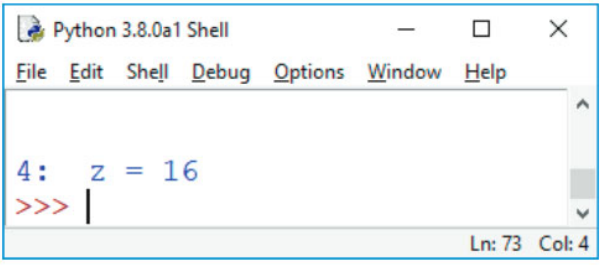
```
z=f22n(a)
print('4: z =',z)
def f22n(a):
  a=a**2
  return a
  a=a*2
  print(a)
```



Поменяем местами тело функции и вызывающие ее операторы. Но в тело добавим операторы после *return*.

```
def f22n(a):
  a=a**2
  return a
  a=a*2
  print(a)
a=4
z=f22n(a)
print('4:  z =',z)
```

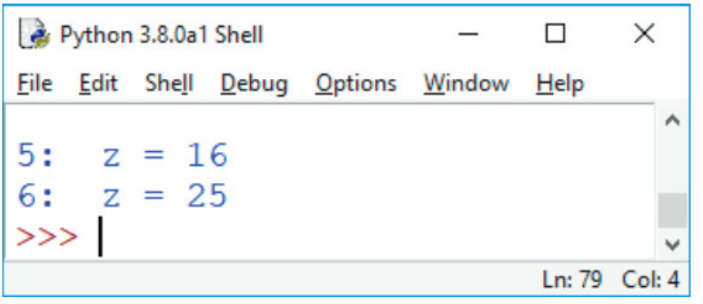
Скриншот показывает, что операторы после *return*, естественно, не выполняются, хотя они набраны с таким же отступом, как операторы в теле функции.



Следующий пример демонстрирует тот факт, что в программе можно определить больше одной функции, но вызывать каждую можно только после ее определения. Тело функции fn1 записано

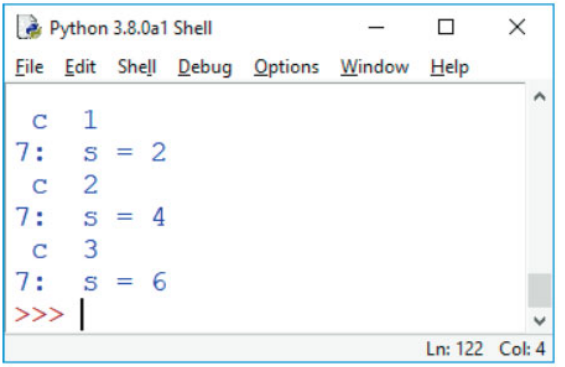
самым первым, далее – операторы, вызывающие ее. Затем следует тело функции fn2 и последним – ее вызов.

```
def fn1(a):
  a=a**2
  return a
b=4
z=fn1(b)
print('5:  z =',z)
def fn2(a):
  a=a**2
  return a
c=5
z=fn2(c)
print('6:  z =',z)
```



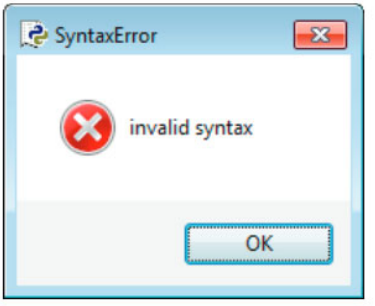
При определении функции может быть задано некоторое фиксированное количество параметров. Приведем два примера, в первом из которых два параметра, а во втором – три. Первая программа вычисляет площадь нескольких прямоугольников, у которых одна сторона фиксирована, а вторая изменяется.

```
def fn3(a,b=2.): # b – не обязательный аргумент
  s=a*b
  return s
for c in range(1,4):
  print(' c ',c)
  z=fn3(c)
  print('7:  s =',z)
```



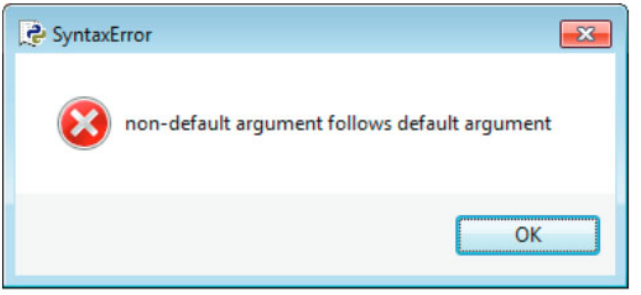
Приведем пример двух текстов функций, в которых записана функция с двумя параметрами. Первому параметру задано значение по умолчанию, а второму – нет. В этом случае выведется сообщение об ошибке.

```
#Вариант 1.
def fn3(a=2,b):
  s=a*b
  return s
for c in range(1,4):
  print(' c ',c)
  z=fn3( ,c)
  print('8: s =',z)
```



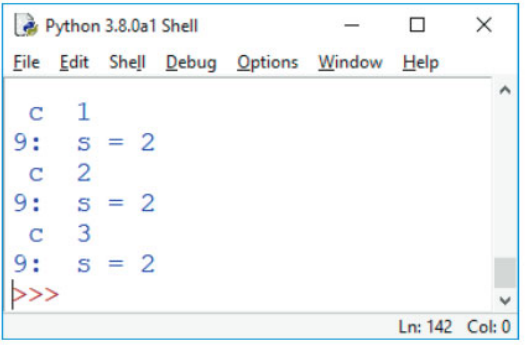

```
#Вариант 2.
def fn3(a=2,b):
  s=a*b
```

```
    return s
for c in range(1,4):
    print(' c ',c)
    z=fn3(c) #  было   z=fn3( ,c)
    print('8:  s =',z)
```



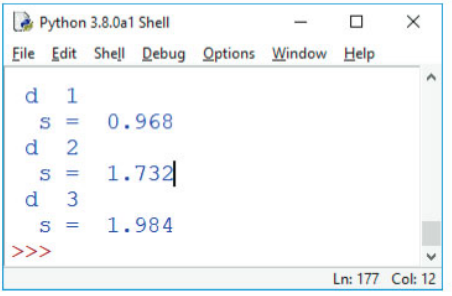
Если же присвоить начальное значение обоим формальным параметрам, то программа должна выдавать всегда один и тот же результат.

```
def fn4(a=2,b=1):
    s=a*b
    return s
for c in range(1,4):
    print(' c ',c)
    z=fn4()
    print('9:  s =',z)
```



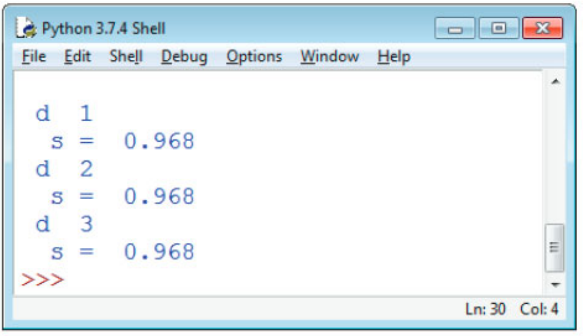
В следующем примере функция имеет три формальных параметра (значения двух из них зафиксировано). В нем вычисляется значение площади треугольника по формуле Герона. В программе используется встроенная функция вычисления корня квадратного, для чего надо подключить (импортировать) модуль *math*. Для более красивого вывода использован метод *format*.

```
import math
def fn5(a,b=2,c=2):
  p=(a+b+c)/2
  s=math.sqrt(p*(p-a)*(p-b)*(p-c))
  return s
for d in range(1,4):
  print(' d ',d)
  z=fn5(d)
  print('  s = {0:6.3f}'.format(z))
```



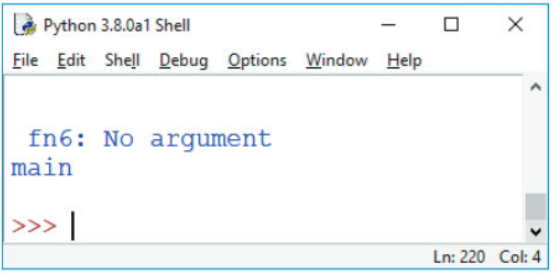
В следующем фрагменте показано, что если при вызове функции задать только один первый параметр, то все три раза получим одно и то же число.

```
import math
def fn5(a,b=2,c=2):
  p=(a+b+c)/2
  s=math.sqrt(p*(p-a)*(p-b)*(p-c))
  return s
for d in range(1,4):
  print(' d ',d)
  z=fn5(a=1)  # a – фиксировано, все аргументы одинаковы
  print('  s = {0:6.3f}'.format(z))
```



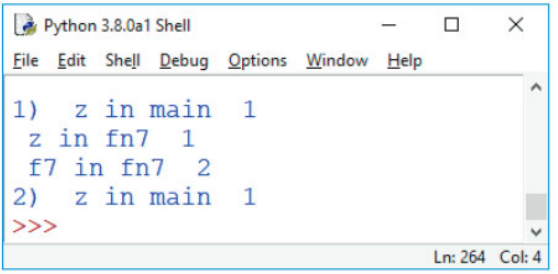
Покажем, что в Питоне можно определить функцию без аргументов.

```
def fn6():
  print(' fn6: No argument ')
fn6()
print('main \n')
```



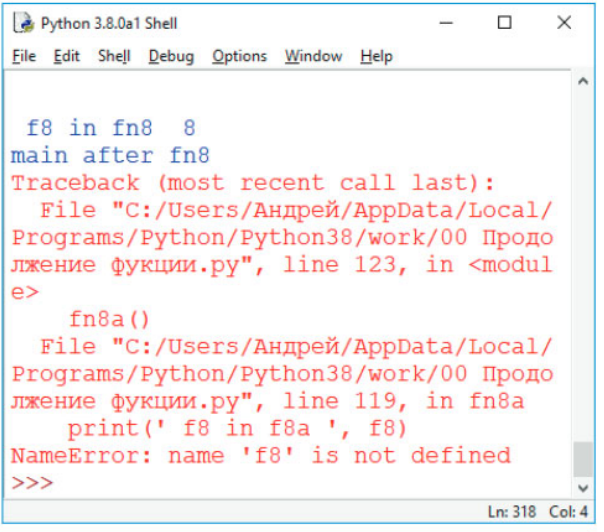
В функции без аргументов можно использовать значения переменных, заданных вне нее, в основном тексте, вне функций. Это демонстрирует следующий пример.

```
def fn7():
  print(' z in fn7 ', z)
  f7=z*2
  print(' f7 in fn7 ', f7)
z=1
print('1)  z in main ',z)
fn7()
print('2)  z in main ',z)
```



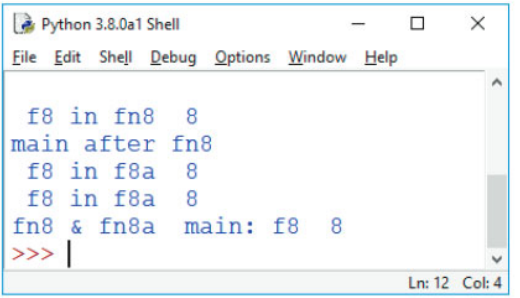
А можно ли использовать значения переменных, определенных в другой функции? Рассмотрим этот вопрос.

```
def fn8():
  f8=8
  print(‘ f8 in fn8 ‘, f8)
def fn8a():
  print(‘ f8 in f8a ‘, f8)
fn8()
print(‘main after fn8’)
fn8a()
print(‘fn8 & fn8a  main ‘)
```



Как видим, переменные, определенные в одной функции, нельзя в целом использовать в другой. Также нельзя даже напечатать значение переменной в функции, если она определена в какой-то другой функции. То есть если изменить оператор print(‘main after fn8’) на такой print(‘main after fn8’,f8), будет такое же сообщение об ошибке: ‘*NameError: name ‘f8’ is not defined*’. Приведенные примеры приводят к выводу о том, что справедливо правило: «Переменные, определенные в функциях, являются локальными». Это значит, что они могут использоваться только в той функции, где определены. Но для Питона, как часто бывает в жизни, есть исключения из этого правила. Такое исключение демонстрируется примером с функцией fn7, в которой используется значение переменной *z*, объявленной вне этой функции. Просто напечатать значение *z* нельзя. Таким образом, переменная *z* только частично приобретает свойство, противоположное «локальная», которое в программировании принято называть «глобальная». В Питоне для придания переменным таких свойств надо использовать ключевое слово ***global***. Покажем его использование, изменив пример с функцией f8.

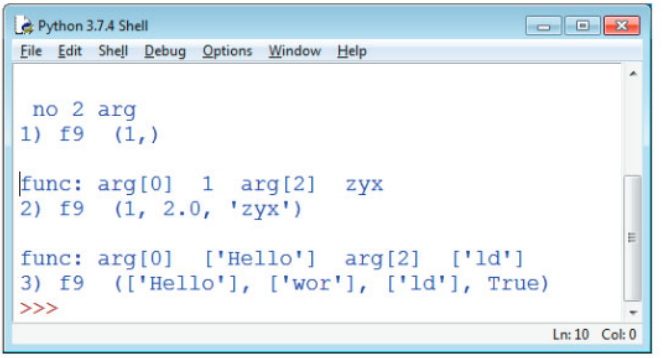
```
def fn8():
#   global f8=8
    global f8
    f8=8
    print(‘ f8 in fn8 ‘, f8)
def fn8a():
    print(‘ f8 in f8a ‘, f8)
    a=f8
    print(‘ f8 in f8a ‘, f8)
fn8()
print(‘main after fn8’)
fn8a()
print(‘fn8 & fn8a  main: f8 ‘, f8)
```



В рассмотренных ранее примерах использовались именные функции с фиксированным количеством аргументов. В противоположность именным функциям в Питоне можно определять анонимные функции, используя ключевое слово (инструкцию) ***lambda***. Но их мы рассмотрим далее. А сейчас перейдем к изучению именных функций с неопределенным количеством аргументов. Они могут быть позиционными и именованными. В примере с функцией f9 она имеет неопределенное количество аргументов. К ней выполнено обращение два раза: с одним и тремя аргументами. Во втором и третьем случаях аргументы f9 имеют разный тип.

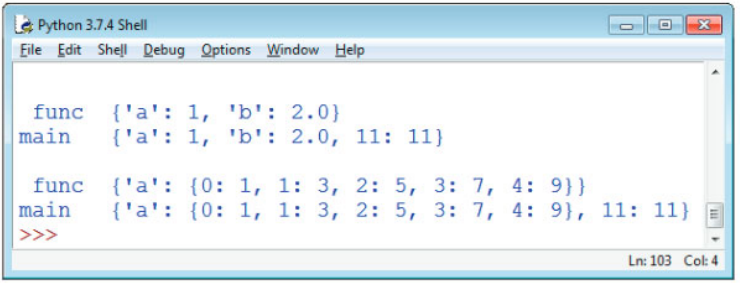
```
def func(*arg):
# NO   arg[0]=arg[0]*2
  if ( len(arg) > 2 ):
    print('func: arg[0] ', arg[0], ' arg[2] ', arg[2] )
  else:
    print(' no 2 arg')
  return arg
f9=func(1)
print('1) f9 ', f9, '\n')
a1=1; a2=2.0
f9=func(a1, a2, 'zyx')
print('2) f9 ', f9, '\n')
f9=func(['Hello'], ['wor'],['ld'], True)
print('3) f9 ', f9)
```

Задавая у функции неопределенное количество позиционных аргументов, мы получаем объект Питона, называемый кортеж (*tuple*). Напомним, что его значения нельзя изменять. К его элементам можно обращаться как к элементам списка, по номеру.



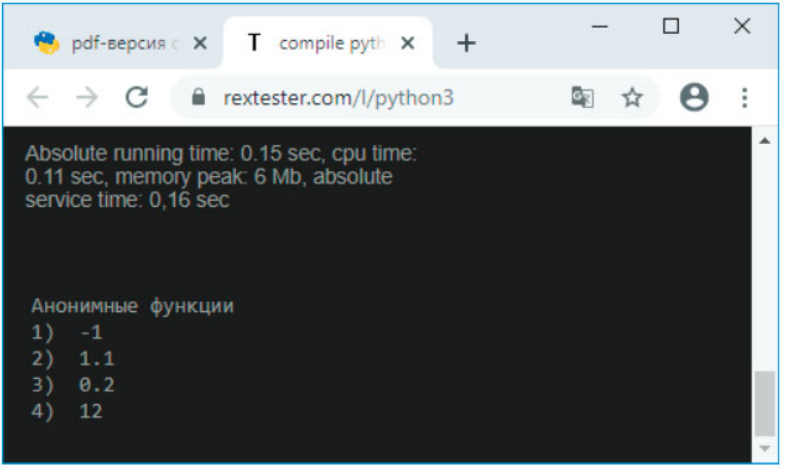
Функции можно задавать произвольное (переменное) количество именованных аргументов. Тогда такой аргумент представляет собой словарь (*dict*), и перед ним ставится два знака звездочка (**).

```
def func(**arg):
  print(‘ func ‘,arg)
  arg[11]=11
  return(arg)
fzz1=func(a=1,b=2.0)
print(‘main ‘, fzz1, ‘\n’)
dic={i: 2*i+1 for i in range(5) }
fzz2=func(a=dic)
print(‘main ‘, fzz2)
```



Здесь закончим рассмотрение именных функций и приведем пример программы, использующей другой тип – анонимные функции. Для их определения требуется инструкция *lambda*.

```
# 1)
print(«\nАнонимные функции»)
fun = lambda a, b: a*b
lmbd1=fun(1, -1)
print('1) ', lmbd1)
# 2)
print( "2) ",(lambda a, b: a*b)(1,1.1) )
# 3)
lmbd2=fun(1, 0.2)
print('3) ', lmbd2)
# 4)
print( "4) ",(lambda a, b: a+b)('1', '2') )
```



Вновь вспомним, что в Интернете доступно множество источников, используя которые можно компилировать и исполнять код программ на разных языках программирования. Последний фрагмент программы выполнялся на ресурсе с адресом https://rextester.com/l/python3.

В разных источниках отмечаются такие особенности анонимных функций: не требуется использовать *return*, они выполняются быстрее, задаются одним выражением. Их можно использовать также и в режиме оболочки. Отметим, что именные функции можно использовать в режиме оболочки.

Экзаменационные темы

1. Создание функций в Питоне.
2. Вызов для выполнения функций в Питоне.
3. Аргументы функций в Питоне.
4. Lambda функции в Питоне.

ЗАКЛЮЧЕНИЕ

Язык программирования Питон имеет множество синтаксических конструкций. Использование многих из них разбираются в учебнике. Подробно разобраны методы и средства поддерживаемых языком типов данных, как однозначные, так и множественные. Приведена информация о том, как разрабатывать собственные функции.

Книга может быть использована для начального изучения языка. Но хорошим подспорьем в этом деле будет опыт разработки программ на другом языке программирования. Даже в объемном издании трудно описать все возможности языка программирования Питон. Авторы планируют подготовить вторую часть учебника, в которой будет описано объектно-ориентированное программирование на этом языке.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Мусин Д. Самоучитель Python. Выпуск 0.2. pythonworld.ru [Электронный ресурс]: URL: https://pythonworld.ru/uploads/pythonworldru.pdf (дата обращения: 10.12.2019).

2. Любанович Б. Простой Python. Современный стиль программирования. СПб.: Питер, 2016. 480 с.: ил.

3. Сысоева М.В., Сысоев И.В. Программирование для нормальных с нуля на языке Python: учебник: в 2-х ч. Ч. 1 / отв. ред. В.Л. Черный. М.: Базальт СПО; МАКС Пресс, 2018. 176 с.

4. Хахаев И.А. Практикум по алгоритмизации и программированию на Python. М.: Альт Линукс, 2010. 126 с.: ил. (Библиотека ALT Linux).

5. Федоров Д.Ю. Программирование на языке высокого уровня Python: учеб. пособие для прикладного бакалавриата. М.: Юрайт, 2017. 126 с. (Серия: Бакалавр. Прикладной курс).

6. The Making of Python. A Conversation with Guido van Rossum. Part I by Bill Venners. [Электронный ресурс]: Copyright © 1996–2018 Artima, Inc. URL: www.artima.com/intv/pythonP.html (дата обращения: 10.08.2019).

7. Питон // Свободная энциклопедия «Википедия» [Электронный ресурс]: Wikipedia® Wikimedia Foundation, Inc. URL: ru.wikipedia.org/wiki/Python (дата обращения: 10.08.2019).

8. История языка// Свободная энциклопедия «Википедия» [Электронный ресурс]: Wikipedia® Wikimedia Foundation, Inc. URL: ru.wikipedia.org/wiki/История_языка_программирования_Python (дата обращения: 10.08.2019).

9. Летающий цирк Монти Пайтона // Свободная энциклопедия «Википедия» [Электронный ресурс]: Wikipedia® Wikimedia Foundation, Inc. URL: ru.wikipedia.org/wiki/Летающий_цирк_Монти_Пайтона (дата обращения: 10.08.2019).

10. The Python Logo. Copyright ©2001-2019. Python Software Foundation. URL: www.python.org/community/logos/ (дата обращения: 10.08.2019).

11. IDLE. Copyright ©2001–2019. Python Software Foundation. URL: https://docs.python.org/3.8/library/idle.html (дата обращения: 10.08.2019).

12. Python Software Foundation License // Свободная энциклопедия «Википедия» [Электронный ресурс]: Wikipedia®

Wikimedia Foundation, Inc. URL: ru.wikipedia.org/wiki/Python_Software_Foundation_License (дата обращения: 10.08.2019).

13. PEP 8 – руководство по написанию кода на Python. © 2012–2017 Python 3 для начинающих. URL pythonworld.ru/osnovy/pep-8-rukovodstvo-po-napisaniyu-koda-na-python.html (дата обращения: 10.08.2019).

14. TIOBE Index for August 2019. URL tiobe. https://tiobe.com/tiobe-index/ (дата обращения: 20.12.2019).

15. IDLE // Свободная энциклопедия «Википедия» [Электронный ресурс]: Wikipedia® Wikimedia Foundation, Inc. URL: ru.wikipedia.org/wiki/IDLE (дата обращения: 10.08.2019).

16. IDLE. Integrated DeveLopment Environment. Copyright © 2012 Try Objective-c. URL: http://www.tryobj.com/27-exam_5kyu.html (дата обращения: 10.08.2019).

17. Tkinter // Свободная энциклопедия «Википедия» [Электронный ресурс]: Wikipedia® Wikimedia Foundation, Inc. URL: ru.wikipedia.org/wiki/Tkinter (дата обращения: 10.08.2019).

18. Стандартная общественная лицензия GNU (GPL). Copyright © Free Software Foundation, Inc. URL: https://www.gnu.org/licenses/gpl-3.0 (дата обращения: 10.08.2019).

19. Brian_Kernighan_Dennis_Ritchie-The_C_Programming_Language-RU.pdf. URL https://www.r-5.org/files/books/computers/languages/c/kr/Brian_Kernighan_Dennis_Ritchie-The_C_Programming_Language-RU.pdf (дата обращения: 10.08.2019).