

ОГЛАВЛЕНИЕ

| | |
|---|----|
| ВВЕДЕНИЕ | 5 |
| 1. УСТАНОВКА И ЗНАКОМСТВО С ЯЗЫКОМ ПРОГРАММИРОВАНИЯ Python | 6 |
| 1.1. Начальные сведения о Python | 6 |
| 1.2. Пример написания простейшей программы в среде разработке IDLE..... | 10 |
| 1.3 Лабораторная работа 1 «Установка и знакомство с языком программирования Python» | 12 |
| 2. ПРОСТЕЙШАЯ ПРОГРАММА И ОСНОВЫ ВВОДА-ВЫВОДА ИНФОРМАЦИИ..... | 14 |
| 2.1. Основы ввода/вывода информации в Python..... | 14 |
| 2.2. Пример написания программы в среде разработке JetBrains PyCharm Community Edition..... | 17 |
| 2.3. Лабораторная работа 2 «Простейшая программа и основы ввода-вывода информации в Python» | 20 |
| 3. ПЕРЕМЕННЫЕ, ПРОСТЫЕ ТИПЫ ДАННЫХ И ОПЕРАЦИИ НАД НИМИ..... | 23 |
| 3.1. Базовый синтаксис языка Python: переменные, простые типы данных и операции над ними..... | 23 |
| 3.2. Лабораторная работа 3 «Переменные, простые типы данных и операции над ними» | 28 |
| 4. УСЛОВНЫЕ КОНСТРУКЦИИ PYTHON..... | 32 |
| 4.1. Базовый синтаксис языка Python: условные конструкции | 32 |
| 4.2. Лабораторная работа 4 «Условные конструкции Python» | 37 |
| 5. ЦИКЛИЧЕСКИЕ КОНСТРУКЦИИ. ИТЕРАЦИОННЫЕ АЛГОРИТМЫ | 39 |
| 5.1. Цикл While..... | 39 |
| 5.2. Цикл For | 42 |
| 5.3. Лабораторная работа 5. Часть 1. «Циклические конструкции. Итерационные алгоритмы. Цикл <i>while</i> » | 43 |
| 5.4. Лабораторная работа 5. Часть 2. «Циклические конструкции. Итерационные алгоритмы. Цикл <i>for</i> » | 45 |
| 6. ФУНКЦИИ. РЕКУРСИВНЫЕ АЛГОРИТМЫ | 47 |
| 6.1. Функции в языке Python. Рекурсивные алгоритмы. Модули..... | 47 |

| | |
|---|----|
| 6.2. Пример выполнения задания для вычисления элементов ряда Фибоначчи..... | 50 |
| 6.3. Пример выполнения задания с использованием рекурсивного подхода..... | 51 |
| 6.4. Лабораторная работа 6 «Функции. Рекурсивные алгоритмы» | 52 |
| 7. МОДУЛИ И ПАКЕТЫ..... | 56 |
| 7.1. Модули и пакеты в Python..... | 56 |
| 7.2. Лабораторная работа 7 «Модули и пакеты» | 57 |
| 8. СТРОКИ И ОБРАБОТКА ТЕКСТОВОЙ ИНФОРМАЦИИ | 58 |
| 8.1. Строки в Python и обработка текстовой информации..... | 58 |
| 8.2. Лабораторная работа 8 «Строки и обработка текстовой информации»..... | 62 |
| 9. КОРТЕЖИ, СПИСКИ, СЛОВАРИ И МНОЖЕСТВА..... | 64 |
| 9.1. Кортежи, списки, словари и множества в Python | 64 |
| 9.2. Пример выполнения индивидуального задания | 70 |
| 9.3. Лабораторная работа 9 «Кортежи, списки, словари и множества»..... | 72 |
| БИБЛИОГРАФИЧЕСКИЙ СПИСОК | 74 |
| ПРИЛОЖЕНИЕ А «Разветвляющиеся алгоритмы»..... | 75 |
| ПРИЛОЖЕНИЕ В «Циклические алгоритмы»..... | 77 |
| ПРИЛОЖЕНИЕ С «Строки» | 82 |
| ПРИЛОЖЕНИЕ D «Списки»..... | 86 |
| ПРИЛОЖЕНИЕ E «Матрицы» | 89 |



*Best of LUCK with it, and remember
to HAVE FUN while you're learning :)
Victor Ivanchenko*

ВВЕДЕНИЕ

Развитие современных информационных технологий напрямую связано с развитием языков программирования. *Python* – современный мощный высокоуровневый язык программирования с поддержкой объектно-ориентированного подхода. Легкость в изучении, удобочитаемость, скорость разработки – далеко не полный перечень достоинств, делающих *Python* таким популярным. Такие крупные компании как *Google*, *Yandex*, *Intel*, *HP*, *IBM*, *NASA* используют в своих разработках *Python*.

Учебно-методическое пособие содержит теоретический материал по основам программирования на языке *Python*, перечень лабораторных работ с рекомендациями по их выполнению, примерами и контрольными вопросами по каждому тематическому разделу.

В пособии представлены следующие тематические блоки:

- выбор среды разработки и создание программ;
- переменные, типы данных, ввод/вывод информации;
- управляющие конструкции;
- обработка исключительных ситуаций;
- пользовательские функции, пакеты, модули;
- строки и обработка текстовой информации;
- списки, кортежи, словари.



We hope you enjoy working with Python!

1. УСТАНОВКА И ЗНАКОМСТВО С ЯЗЫКОМ ПРОГРАММИРОВАНИЯ Python

1.1. Начальные сведения о Python

Язык программирования *Python* – современный мощный высокоуровневый язык программирования, созданный голландским программистом Гвидо Ван Россумом и представленный в 1991 году. Своему названию язык обязан популярному комедийному телесериалу «Воздушный цирк Монти Пайтона», который транслировался по каналу *BBC*, что неизбежно добавляет юмора в примеры кода на языке *Python*.

Python испытал на себе влияние таких языков программирования, как *Java*, *C*, *C++* (как отмечает сам автор – он использовал наиболее непротиворечивые конструкции из *C*, чтобы не вызвать неприязнь у *C*-программистов к новому языку. В то же время *Python* оказал влияние на язык программирования *Ruby*.

Официальный сайт языка программирования *Python* – <http://python.org>.

Python можно рассматривать как

- интерпретируемый язык: исходный код на *Python* не компилируется в машинный код, а выполняется непосредственно с помощью специальной программы-интерпретатора;

- интерактивный язык: можно писать код прямо в оболочке интерпретатора и вводить новые команды по мере выполнения предыдущих;

- объектно-ориентированный язык: *Python* поддерживает принципы объектно-ориентированного программирования, которые подразумевают инкапсуляцию кода в особые структуры, именуемые объектами.

Особенности языка *Python*

- легкий для изучения: в *Python* немного ключевых слов, простая структура и четко определенный синтаксис, что позволяет научиться основам языка за достаточно короткое время;

- легко читаемый: блоки кода в *Python* выделяются при помощи отступов, что совместно с ключевыми словами, взятыми из английского языка, значительно облегчает его чтение;
- простота обслуживания кода;
- наличие широкой кросс-платформенной библиотеки;
- наличие интерактивного режима позволяет «на лету» тестировать нужные участки кода;
- портативность: *Python* без проблем запускается на разных платформах, сохраняя при этом одинаковый интерфейс;
- расширяемость: при необходимости в *Python* можно внедрять низкоуровневые модули, написанные на других языках программирования, для наиболее гибкого решения задач;
- работа с базами данных: в стандартной библиотеке *Python* можно найти модули для работы с большинством баз данных;
- создание *GUI* (графического интерфейса пользователя): предусмотрена возможность создания *GUI* приложений с самым широким спектром пользовательских настроек.

Недостатки *Python*

Скорость выполнения программ несколько ниже по сравнению с компилируемыми языками (*C* или *C++*), поскольку *Python* транслирует инструкции исходного программного кода в байт-код, а затем интерпретирует этот байт-код. Байт-код обеспечивает переносимость программ, поскольку это платформонезависимый формат. Как результат, имеет место преимущество в скорости разработки с потерей скорости выполнения.

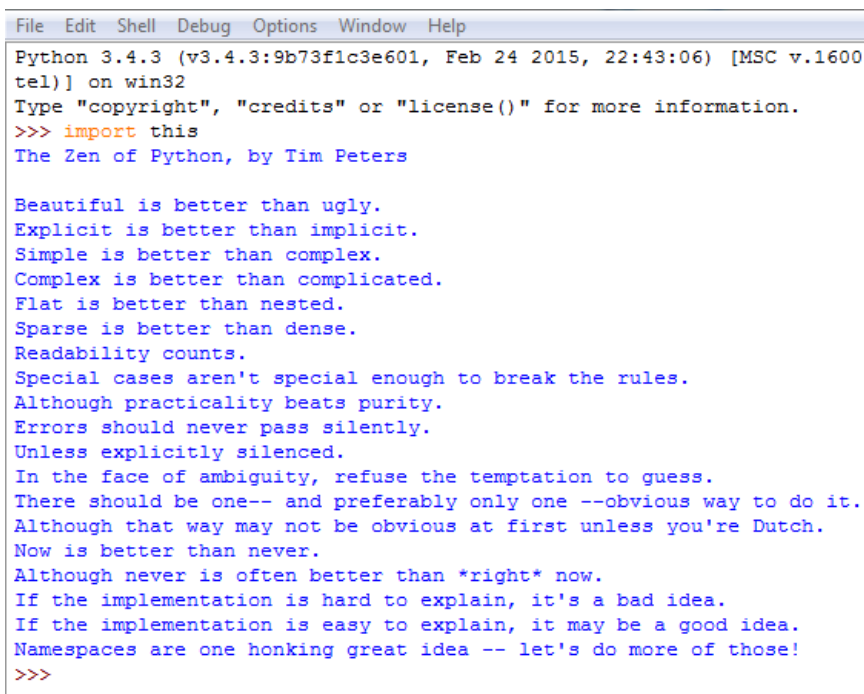
Кто в наше время использует *Python*?

- Компания *Google* оплачивает труд создателя *Python* и использует язык в поисковых системах;
- сервис *YouTube*;
- компания *Yandex* в ряде сервисов;
- программа *BitTorrent* написана на *Python*;
- веб-фреймворк *App Engine* от *Google* использует *Python* в качестве прикладного языка программирования;

- *Intel, Cisco, HP, IBM* используют *Python* для тестирования аппаратного обеспечения;
- *NASA* использует язык в научных вычислениях.

Дзен *Python*'а

Если интерпретатору в *Python* дать команду *import this*, то, как показано на рис. 1.1, выведется, так называемый, «*Zen of Python*», иллюстрирующий идеологию и особенности данного языка. Глубокое понимание дзена (табл. 1.1) приходит тем, кто сможет освоить язык *Python* в полной мере.



```
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import this
The Zen of Python, by Tim Peters

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

Рис. 1.1. *Zen of Python*

Таблица 1.1

Перевод «Zen of Python»

| | Фраза | Перевод |
|-----|---|--|
| 1. | <i>Beautiful is better than ugly</i> | Красивое лучше уродливого |
| 2. | <i>Explicit is better than implicit</i> | Явное лучше неявного |
| 3. | <i>Simple is better than complex</i> | Простое лучше сложного |
| 4. | <i>Complex is better than complicated</i> | Сложное лучше усложненного |
| 5. | <i>Flat is better than nested</i> | Плоское лучше вложенного |
| 6. | <i>Sparse is better than dense</i> | Разреженное лучше плотного |
| 7. | <i>Readability counts</i> | Удобочитаемость важна |
| 8. | <i>Special cases aren't special enough to break the rules</i> | Частные случаи не настолько существенны, чтобы нарушать правила |
| 9. | <i>Although practicality beats purity</i> | Однако практичность важнее чистоты |
| 10. | <i>Errors should never pass silently</i> | Ошибки никогда не должны замалчиваться |
| 11. | <i>Unless explicitly silenced</i> | За исключением замалчивания, которое задано явно |
| 12. | <i>In the face of ambiguity, refuse the temptation to guess</i> | В случае неоднозначности сопротивляйтесь искушению угадать |
| 13. | <i>There should be one – and preferably only one – obvious way to do it</i> | Должен существовать один – и, желательно, только один – очевидный способ сделать это |
| 14. | <i>Although that way may not be obvious at first unless you're Dutch</i> | Хотя он может быть с первого взгляда не очевиден, если ты не голландец |
| 15. | <i>Now is better than never</i> | Сейчас лучше, чем никогда |
| 16. | <i>Although never is often better than «right» now</i> | Однако, никогда чаще лучше, чем прямо сейчас |
| 17. | <i>If the implementation is hard to explain, it's a bad idea</i> | Если реализацию сложно объяснить – это плохая идея |
| 18. | <i>If the implementation is easy to explain, it may be a good idea</i> | Если реализацию легко объяснить – это может быть хорошая идея |
| 19. | <i>Namespaces are one honking great idea – let's do more of those!</i> | Пространства имен – прекрасная идея, давайте делать их больше! |

1.2. Пример написания простейшей программы в среде разработке *IDLE*

После загрузки и установки *Python* необходимо открыть *IDLE*, среда разработки на языке *Python* поставляется вместе с дистрибутивом (рис. 1.2).

Здесь и далее будут приводиться примеры под ОС *Windows 7*.

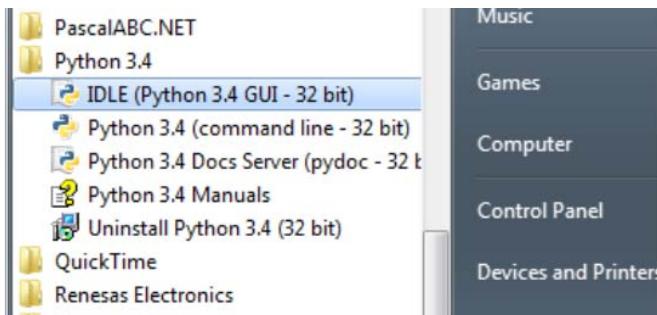


Рис. 1.2. Запуск *IDLE*

После запуска *IDLE* (изначально запускается в интерактивном режиме) можно начинать писать первую программу.

Рассмотрим создание программы, выводящей на экран приветственное сообщение «*Hello, students! :))*». Для этого необходимо использовать инструкцию `print("Hello, students! :))")`. Код вводится в *IDLE*, результат отображается после нажатия *Enter*. Результат работы программы приведен на рис. 1.3.

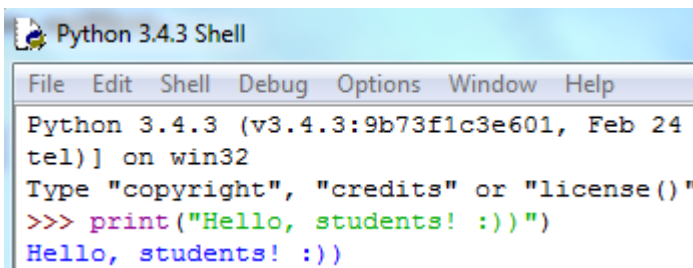


Рис. 1.3. Ввод-вывод сообщения

Интерактивный режим удобен, но крайне ограничен по своим возможностям. В основном, программный код создается в сценарном режиме, сохраняется в файл и запускается после сохранения этого файла.

Для того чтобы создать новое окно и перейти в сценарный режим, в интерактивном режиме *IDLE* необходимо выбрать *File* → *New File* или нажать *Ctrl + N* (рис. 1.4).

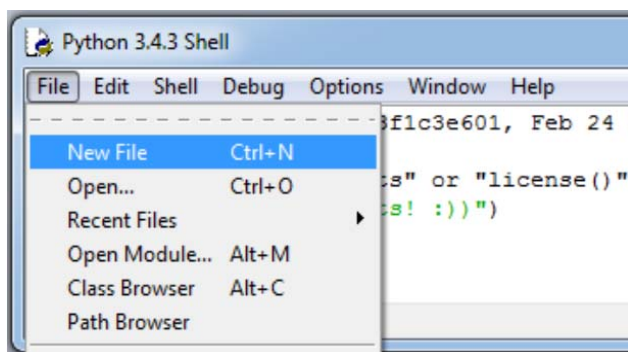


Рис. 1.4. Создание нового окна

В качестве примера рассмотрим программу, которая спрашивает имя пользователя и здоровается, обращая к нему по имени (рис. 1.5). Первая строка печатает вопрос («*What is your name?*»), ожидает, пока пользователь не напечатает что-нибудь и не нажмет *Enter*, и сохраняет введенное значение в переменной *name*.

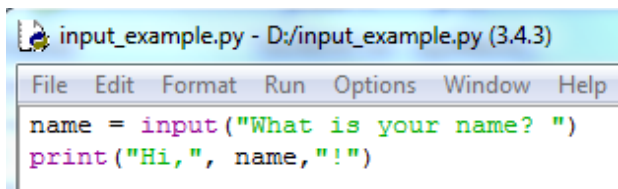
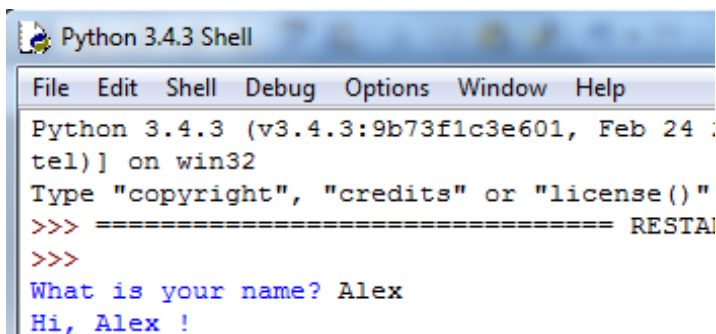


Рис. 1.5. Код программы

Для запуска программы необходимо нажать *F5* или выбрать в меню *IDLE* *Run* → *Run Module*. Перед запуском *IDLE* предложит сохранить файл, после чего программа запустится. Результат работы программы приведен на рис. 1.6.



```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 :
tel)] on win32
Type "copyright", "credits" or "license()"
>>> ===== RESTA
>>>
What is your name? Alex
Hi, Alex !
```

Рис. 1.6. Результат запуска программы



Поздравляем! Вы научились писать простейшие программы, а также познакомились со средой разработки *IDLE*!

1.3. Лабораторная работа 1 «Установка и знакомство с языком программирования *Python*»

Цель работы: установить интегрированную среду разработки Python и научиться писать, редактировать, сохранять и запускать программы на Python.

Основное задание

1. Установить интегрированную среду разработки *Python* (рекомендуется скачать установщик с официального сайта www.python.org).
2. Ознакомиться с базовой документацией по языку программирования *Python*.
3. Протестировать среду разработки *IDLE* в двух режимах: в интерактивном и сценарном посредством разработки приложения, образец которого приведен в разделе «Пример написания простейшей программы в среде разработке *IDLE*».
4. Изучить основные горячие клавиши для последующей быстрой разработки программ в сценарном режиме.

Индивидуальное задание

Написать простейшую программу, которая будет выводить на экран любимый афоризм студента или его девиз, а на отдельной строке информацию об авторе, например, инициалы студента.

Требования к выполнению

Программа должна обязательно быть снабжена комментариями, в которых необходимо указать краткое назначение программы, ее версию, Ф.И.О. разработчика, номер группы и дату разработки.

Контрольные вопросы

1. Что такое алгоритм?
2. Что такое язык программирования?
3. Что такое компьютерная программа?
4. Почему можно рассматривать *Python* как интерпретируемый, интерактивный и объектно-ориентированный язык программирования?
5. В чем особенности *Python*?
6. С чем связана скорость выполнения программ на *Python*?
7. Что представляет собой язык программирования *Python* и чем он интересен?
8. Что такое *Zen of Python*?
9. Что такое *IDLE*?
10. В чем отличия работы в интерактивном и сценарном режимах среды разработки *IDLE*?
11. Зачем необходимо использовать комментарии в программе?
12. Как вывести сообщение на *Python*?
13. Какие типы комментариев существуют в *Python*?

2. ПРОСТЕЙШАЯ ПРОГРАММА И ОСНОВЫ ВВОДА-ВЫВОДА ИНФОРМАЦИИ

2.1. Основы ввода/вывода информации в *Python*

Escape-последовательности

Для достижения большей гибкости в отображении текста на экране в строки можно вставлять специальные *escape-последовательности*, которые не отображаются на экране и всегда начинаются с символа правого слеша ('\'). Наиболее часто используемые *escape-последовательности* приведены в табл. 2.1.

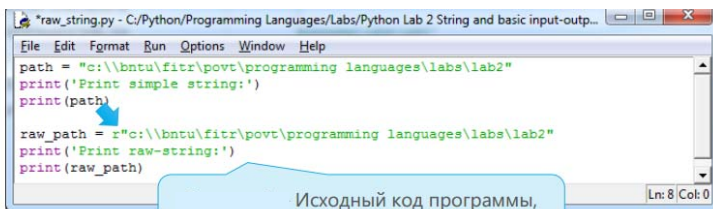
Таблица 2.1

Escape-последовательности

| Обозначение | Описание |
|-------------|--|
| \t | горизонтальный табулятор |
| \n | вертикальный табулятор |
| \s | пробел |
| \b | удаление предыдущего |
| \x | символ <i>x</i> |
| \n | новая строка |
| \r | возврат каретки |
| \a | вызов одинарного сигнала системного динамика |
| \' | вывод одинарной кавычки – апострофа |
| \" | вывод двойной кавычки |
| \\ | вывод правого слеша |

Raw-строки

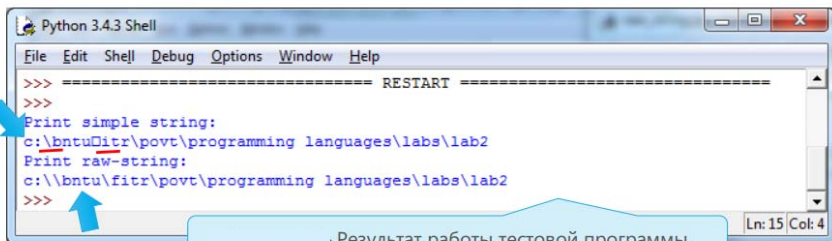
Если требуется вывести строку точно в таком же виде, как она описана в коде, без всевозможных преобразований символов *escape-последовательностей*, достаточно перед написанием самой строки поставить префикс в виде символа *r* (*raw* – запись/строка). Данный тип строки не преобразует слешы. Сравнение результатов вывода строк и *raw*-строк приводится на рис. 2.1.



```
File Edit Format Run Options Window Help
path = "c:\\bntu\\fitr\\povt\\programming languages\\labs\\lab2"
print('Print simple string:')
print(path)

raw_path = r"c:\\bntu\\fitr\\povt\\programming languages\\labs\\lab2"
print('Print raw-string:')
print(raw_path)
```

Исходный код программы,
которая тестирует простые и raw-строки



```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
>>> ===== RESTART =====
>>>
>>> Print simple string:
c:\\bntu\\fitr\\povt\\programming languages\\labs\\lab2
Print raw-string:
c:\\bntu\\fitr\\povt\\programming languages\\labs\\lab2
>>>
```

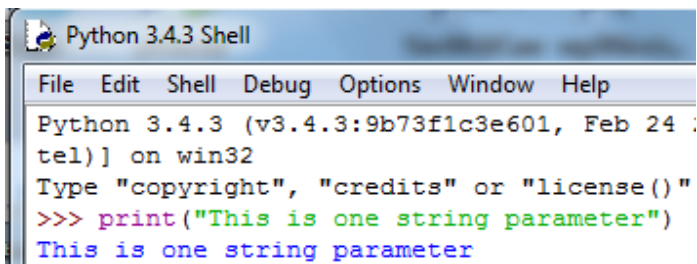
Результат работы тестовой программы

Рис. 2.1. Результаты вывода с использованием raw-строк

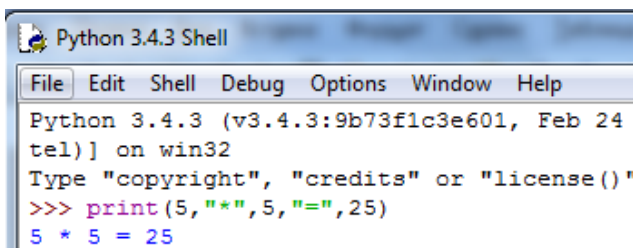
Символы в памяти компьютера хранятся с помощью номеров, то есть их числовых кодов. По умолчанию все строки в *Python 2* кодируются с использованием *ASCII*-кодировки. В *Python 3* все строки представляются в *Unicode*-кодировке!

Использование функций `PRINT()` и `INPUT()`

Функция для вывода информации на дисплей `print()`. Стандартная функция `print()` – это базовая функция для организации вывода одного или нескольких значений по умолчанию на консоль (для этого используется модуль `sys.stdout`) или в любой другой указанный поток (*stream*) (рис. 2.2).



```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015) on win32
Type "copyright", "credits" or "license()"
>>> print("This is one string parameter")
This is one string parameter
```



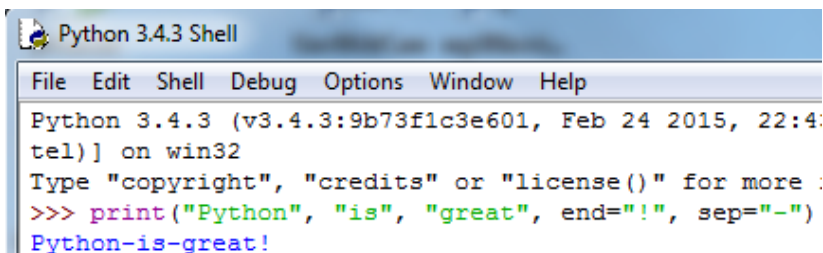
```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015) on win32
Type "copyright", "credits" or "license()"
>>> print(5, "*", 5, "=", 25)
5 * 5 = 25
```

Рис. 2.2. Вывод информации на дисплей `print()`

Дополнительные параметры функции `print()`:

- *sep* – строка или символ, который вставляется между значениями функции, по умолчанию используется пробел;
- *end* – строка, которая подставляется после последнего выводимого функцией значения, по умолчанию – символ новой строки.

Дополнительные параметры идут последними (рис. 2.3).



```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:48) on win32
Type "copyright", "credits" or "license()" for more:
>>> print("Python", "is", "great", end="!", sep="-")
Python-is-great!
```

Рис. 2.3. Пример использования *sep* и *end* в функции `print()`

Функция для ввода данных с клавиатуры `input()`. Стандартная функция `input()` – базовая функция для организации ввода данных. В качестве необязательного параметра она принимает строку-приглашение и возвращает строку, вводимую пользователем.

Основные правила использования функции `input()`:

- строка-приглашение не начинается и не заканчивается с символа новой строки, то есть пользователь должен заблаговременно учесть организацию удобного запроса на ввод данных;
- функция `input()` часто используется для того, чтобы организовать паузу перед закрытием программы для просмотра результата ее выполнения. Синтаксис функции `input()` представлен на рис. 2.4.

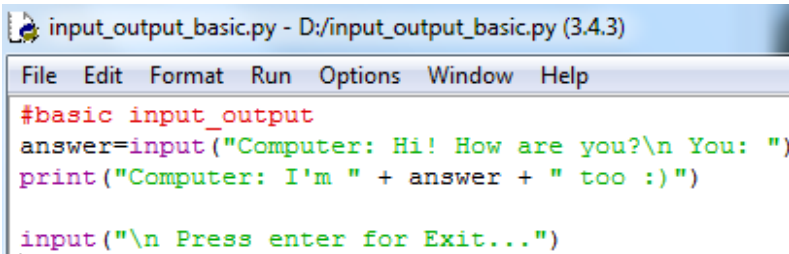


Рис. 2.4. Синтаксис функции `input()`

Результат работы программы при организации диалога между пользователем и компьютерной программой (рис. 2.5).

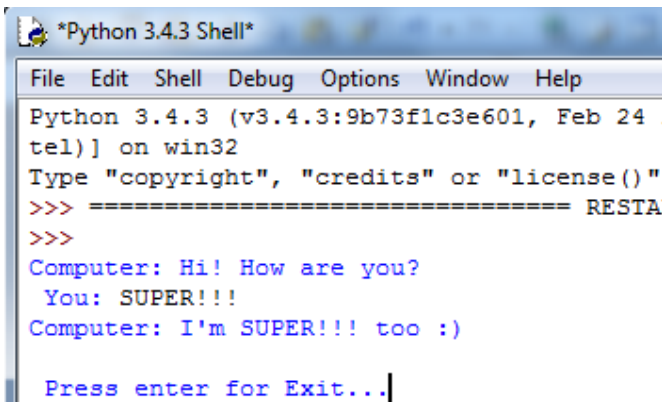


Рис. 2.5. Диалог между пользователем и компьютерной программой

2.2. Пример написания программы в среде разработки *JetBrains PyCharm Community Edition*

Этот раздел посвящается тем, кому не нравится нативная среда разработки *Python IDLE*, или тем, кто стремится овладеть более мощными средствами разработки программ на *Python*.

После установки интерпретатора *Python* и среды разработки *JetBrains PyCharm Community Edition* для ее запуска нужно щелкнуть мышкой на соответствующем ярлыке на рабочем столе или выбрать запуск данной среды из меню «Пуск» (рис. 2.6).

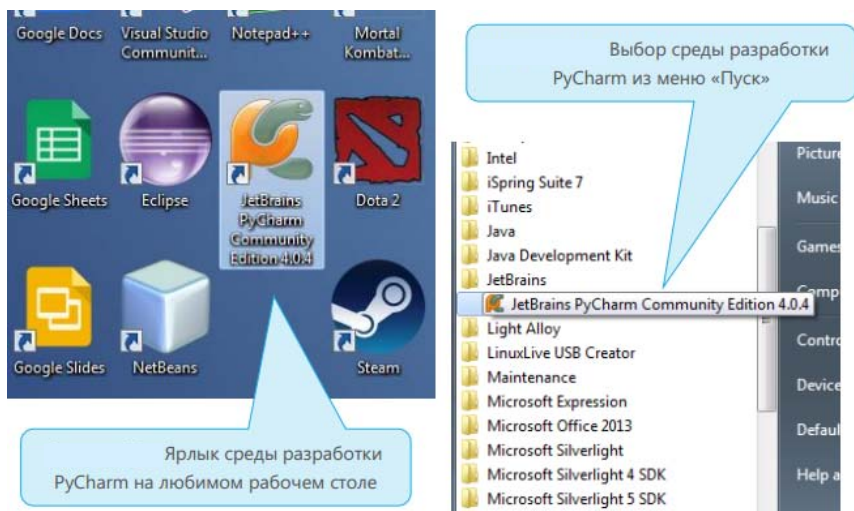


Рис. 2.6. Вызов среды разработки *JetBrains PyCharm Community Edition*

Этапы разработки программы в среде *PyCharm*:

- 1) создание нового проекта «*Create New Project*» (см. рис. 2.6);
- 2) сохранение проекта – в поле «*Location*» (месторасположение) выбрать место на диске, где будет сохранен проект и его имя;
– в поле «*Interpreter*» указать место расположения *Python*,
– нажать кнопку «*Create*»;
- 3) создание файла-скрипта – в появившемся основном окне среды разработки в левой панели «*Project*» на проекте *Lab2Project* щелкнуть правой кнопкой мыши и из контекстного меню и выбрать *New*→*Python File*;
- 4) название файла-скрипта – в появившемся диалоговом окне необходимо ввести имя файла-скрипта и нажать кнопку *OK*;
- 5) написание текста программы – в основном окне файла-скрипта вводится исходный текст программы (рис. 2.7);

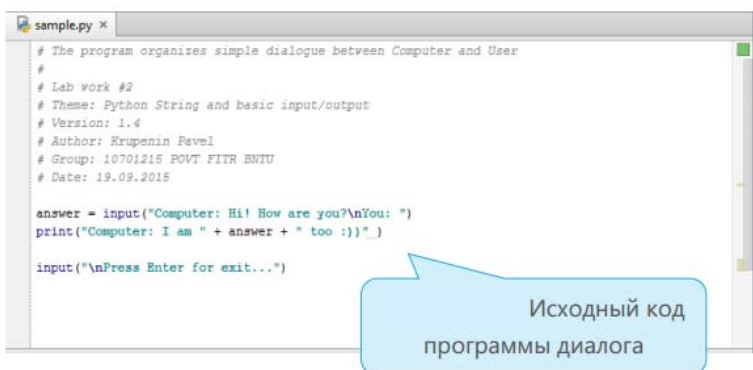


Рис. 2.7. Написание текста программы

- 5) запуск программы – можно использовать несколько способов:
- из главного меню среды разработки выбрать раздел *Run* → *Run 'sample'* или использовать сочетание горячих клавиш *Shift + F10*;
 - нажать зеленую стрелочку: в верхнем правом углу среды разработки или внизу слева в окне просмотра результата интерпретации исходного кода программы (рис. 2.8);
- 6) результат выполнения программы приводится на рис. 2.9.

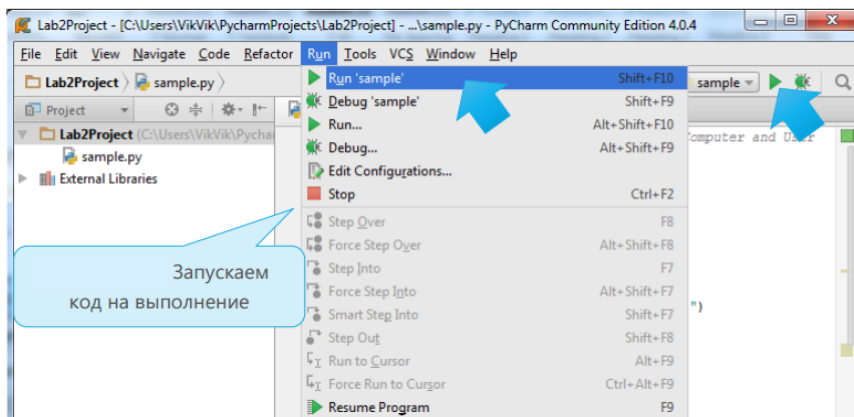


Рис. 2.8. Запуск программы

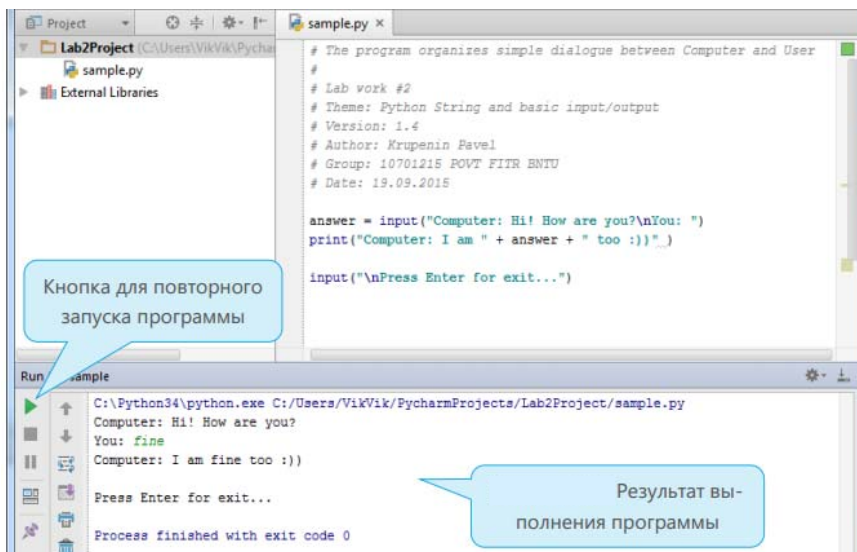


Рис. 2.9. Результат выполнения программы



Поздравляем! Вы научились писать программы с использованием специализированной среды разработки *JetBrains PyCharm Community Edition*.

2.3. Лабораторная работа 2

«Простейшая программа и основы ввода-вывода информации в *Python*»

Цель работы: научиться использовать стандартные функции ввода-вывода *Python* для написания интерактивных программ, организуемых диалог между пользователем и компьютером.

Основное задание

1. Установить специализированную среду разработки *JetBrains PyCharm Community Edition*.
2. В установленной среде разработки создать программу, выводящую на экран адрес расположения заданий для лабораторных работ на сервере и адрес любого электронного ресурса, содержащего

программное обеспечение для разработки на языке программирования *Python*.

3. Разработать программу «*Game Over*», которая выводит соответствующую запись на экран монитора с внушительным видом. Художественное оформление выводимой информации ограничено только фантазией разработчика.

Индивидуальное задание

Написать интерактивную программу, которая будет запрашивать соответствующие данные о студенте (Ф.И.О., дата рождения, адрес (можно только город), любимый род деятельности, любимое блюдо, любимый фильм и так далее), а затем выводить их в табличном виде на экран монитора. Для формирования таблицы можно воспользоваться выводом символов таблицы *ASCII*-кодов с помощью *escape*-последовательностей, но не обязательно. Разработчику дается полный полет фантазий.

Дополнительное задание

Написать программу «*Рифмоплет*», которая, используя дополнительные параметры функции `print()`, будет запрашивать у пользователя несколько рифм и заканчивать ими заранее подготовленное стихотворение.

Требования к выполнению

Программа должна обязательно быть снабжена комментариями на английском языке, в которых необходимо указать краткое предназначение программы, номер лабораторной работы, название лабораторной работы, версию программы, Ф.И.О. разработчика, номер группы и дату разработки.

Контрольные вопросы

1. Какими способами отображаются строки в *Python*?
2. Зачем в *Python* используют тройные кавычки (апострофы)?
3. Зачем нужны *escape*-последовательности?
4. Зачем нужны *raw*-строки и как их записать в коде?
5. Что такое таблица кодировок?
6. Чем отличается *ASCII*-кодировка от *Unicode*-кодировки?

7. Какой стандарт (тип кодировки) используется для кодирования символьной информации в *Python 3*?
8. Опишите синтаксис функции `print()`.
9. Опишите синтаксис функции `input()`.
10. С помощью какой функции в *Python* можно узнать подробную справочную информацию?
11. Предположим, в коде программы используется *escape*-последовательность `'\a'` для создания звука с помощью системного динамика. В каком случае при запуске программы на выполнение будет слышен звук?
12. Описать кратко опыт, который был проведен в Йельском университете США в 1953 и 1973 годах. Каков был результат опыта? Какой можно сделать для себя вывод?

3. ПЕРЕМЕННЫЕ, ПРОСТЫЕ ТИПЫ ДАННЫХ И ОПЕРАЦИИ НАД НИМИ

3.1. Базовый синтаксис языка *Python*: переменные, простые типы данных и операции над ними

Информация, сохраненная в памяти компьютера, может быть разных типов данных. К стандартным типам данных, используемым в *Python*, относятся: число (*Number*); строка (*String*); список (*List*); кортеж (*Tuple*); словарь (*Dictionary*); множество (*Set*).

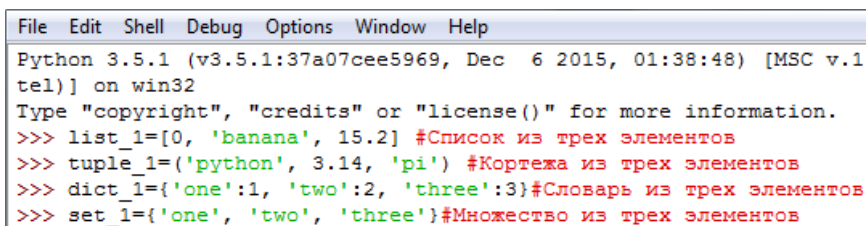
Числовой тип данных в *Python* предназначен для хранения числовых значений и представляет собой *неизменяемый тип данных*, то есть изменение значения числового типа приведет к созданию нового объекта в памяти (и удалению старого).

Для хранения и обработки текстовой информации используется *строковый тип данных*. В *Python* строки могут задаваться следующими способами:

- 1) строка в одинарных кавычках (апострофах);
- 2) строка в двойных кавычках;
- 3) строка в тройных кавычках;

Особенность строки в тройных кавычках состоит в том, что она может занимать в коде несколько строк и при этом выводиться на экран точно в таком же виде, как и вводится!

Списки, кортежи, словари и множества — представляют собой последовательности значений произвольных типов (рис. 3.1).



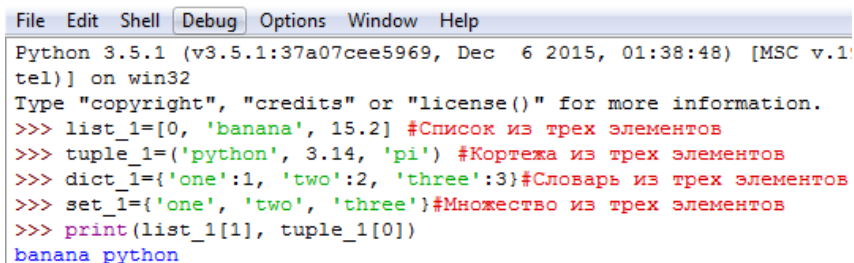
```
File Edit Shell Debug Options Window Help
Python 3.5.1 (v3.5.1:37a07cee5969, Dec 6 2015, 01:38:48) [MSC v.1
tel] on win32
Type "copyright", "credits" or "license()" for more information.
>>> list_1=[0, 'banana', 15.2] #Список из трех элементов
>>> tuple_1=('python', 3.14, 'pi') #Кортежа из трех элементов
>>> dict_1={'one':1, 'two':2, 'three':3}#Словарь из трех элементов
>>> set_1={'one', 'two', 'three'}#Множество из трех элементов
```

Рис. 3.1. Объявление списка, кортежа, словаря и множества

Числа, строки и кортежи представляют собой *неизменяемые типы данных*, то есть изменение значения переменной каждого типа

приведет к созданию нового объекта в памяти (и удалению старого). Списки, множества и словари представляют собой *изменяемые последовательности*.

Обращение к элементам последовательностей осуществляется с помощью квадратных скобок (рис. 3.2).

A screenshot of a Python IDE window. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The title bar reads 'Python 3.5.1 (v3.5.1:37a07cee5969, Dec 6 2015, 01:38:48) [MSC v.1...tel)] on win32'. The main text area contains the following code:

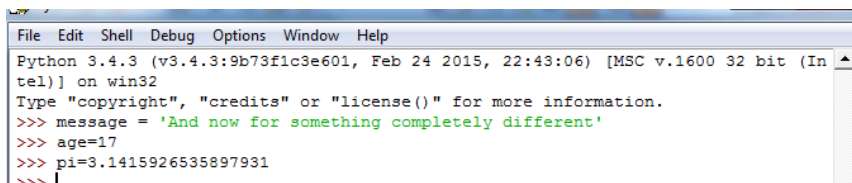
```
Type "copyright", "credits" or "license()" for more information.
>>> list_1=[0, 'banana', 15.2] #Список из трех элементов
>>> tuple_1=('python', 3.14, 'pi') #Кортежа из трех элементов
>>> dict_1={'one':1, 'two':2, 'three':3}#Словарь из трех элементов
>>> set_1={'one', 'two', 'three'}#Множество из трех элементов
>>> print(list_1[1], tuple_1[0])
banana python
```

Рис. 3.2. Вывод элементов списка и кортежа

Переменная – это имя, которое ссылается на значение.

Выражение – комбинация значений, переменных и операторов.

Оператор присваивания (assignment statement) « \leftarrow » создает новые переменные и присваивает им значения (рис. 3.3).

A screenshot of a Python IDE window. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The title bar reads 'Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (Intel)] on win32'. The main text area contains the following code:

```
Type "copyright", "credits" or "license()" for more information.
>>> message = 'And now for something completely different'
>>> age=17
>>> pi=3.1415926535897931
>>>
```

Рис. 3.3. Оператор присваивания

Типы переменных – это типы значений, на которые они ссылаются (рис. 3.4). Для переменных, как правило, используют имена, которые раскрывают их назначение. Имена переменных должны начинаться с буквы, могут содержать буквы, цифры, символы подчеркивания; нежелательно использовать буквы верхнего регистра и нельзя использовать зарезервированные слова (рис. 3.5).

В *Python* имеет место *динамическая типизация*, где любой объект является ссылкой, а типом объекта является то, на что он ссылается. Тип может динамически меняться в процессе выполнения программы, когда ссылка начинает указывать на объект другого типа.

```
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73fic3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> message = 'And now for something completely different'
>>> age=17
>>> pi=3.1415926535897931
>>> type(message)
<class 'str'>
>>> type(age)
<class 'int'>
>>> type(pi)
<class 'float'>
>>> |
```

Рис. 3.4. Типы переменных

```
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73fic3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import keyword
>>> keyword.kwlist
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', '
def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if',
'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'retu
rn', 'try', 'while', 'with', 'yield']
```

Рис. 3.5. Зарезервированные слова в Python

Операторы и операнды

Говоря простым языком, в выражении 5 + 7 числа 5 и 7 называются операндами, знак «+» оператором. В Python существуют следующие *типы операторов*: арифметические операторы, операторы сравнения, операторы присваивания, побитовые операторы, логические операторы, операторы членства и тождественности. В табл. 3.1–3.7 описаны наиболее используемые операторы.

Таблица 3.1

Арифметические операторы

| Арифметические операторы | | | | Описание |
|--------------------------|---|---|---|---|
| + | − | * | / | Сложение, вычитание, умножение и деление. |
| ** | | | | Возведение в степень. |
| % | | | | Деление по модулю – делит левый операнд на правый и возвращает остаток. |
| // | | | | Целочисленное деление операндов. |

Таблица 3.2

Строковые операторы

| Строковые операторы | Описание |
|---------------------|---|
| + | Конкатенация (<i>concatenation</i>) строки. |
| * | Дублирование строки несколько раз. |

Таблица 3.3

Строковые операторы

| Операторы сравнения | Описание |
|---------------------|--|
| == | Проверка равенства двух операндов. |
| != | Проверка неравенства двух операндов. Возвращает ложь, если операнды равны. |
| < | Проверка, больше (меньше, не меньше, не больше) ли левый операнд правого. |
| > | |
| <= | |
| >= | |

Таблица 3.4

Операторы присваивания

| Операторы присваивания | Описание |
|------------------------|---|
| = | Оператор присваивания. |
| += | Присваивание левому операнду значения суммы правого и левого операндов. Например, $x += y$ равносильно: $x = x + y$. |

Таблица 3.5

Логические операторы

| Логические операторы | Описание |
|----------------------|---|
| <i>and</i> | Логическое «И». Возвращает истину, если оба операнда истинны. |
| <i>or</i> | Логическое «ИЛИ». Возвращает истину, если хотя бы один из операндов истинный. |
| <i>not</i> | Логическое «НЕ». Изменяет логическое значение операнда на противоположное. |

Приоритет операторов в *Python*

В табл. 3.6 описан приоритет операторов в *Python*.

Таблица 3.6

Приоритет операторов в *Python*

| Операторы | Описание |
|---|---|
| ** | Возведение в степень. |
| ~ | Комплиментарный оператор. |
| * / % // | Умножение, деление, деление по модулю, целочисленное деление. |
| + - | Сложение, вычитание. |
| >> << | Побитовый сдвиг вправо (влево). |
| & | Бинарное «И». |
| ^ | Бинарные «Исключительное ИЛИ», «ИЛИ». |
| > < >= <= | Операторы сравнения. |
| == != <> | Операторы равенства. |
| = += -= ... | Операторы присваивания. |
| is is not | Тождественные операторы. |
| in not in | Операторы членства. |
| not or and | Логические операторы. |

Преобразование типов данных

Иногда может возникнуть необходимость преобразовать один тип данных в другой. Для этого существуют специальные встроенные функции *Python*. Некоторые из них представлены в табл. 3.7.

Таблица 3.7

Некоторые встроенные функции в *Python*

| Функция | Описание | Пример |
|------------------------|---|-------------------------|
| <i>int(x [,base])</i> | Преобразование <i>x</i> в целое число. | <i>int(12.4) → 12</i> |
| <i>long(x [,base])</i> | Преобразование <i>x</i> в тип <i>long</i> . | <i>long(20) → 20L</i> |
| <i>float(x)</i> | Преобразование <i>x</i> в вещественное число. | <i>float(10) → 10.0</i> |
| <i>str(x)</i> | Преобразование <i>x</i> в строку. | <i>str(10) → "10"</i> |

3.2. Лабораторная работа 3 «Переменные, простые типы данных и операции над ними»

Цель работы: освоить базовый синтаксис языка *Python*, простые типы данных, приобрести навыки создания интерактивных программ с использованием линейных алгоритмов.

Основное задание

1. Разработать интерактивную программу, демонстрирующую работу с простыми типами данных:

| | |
|-----|---|
| 1.1 | Программа должна запрашивать имя пользователя (строка) и почтовый индекс (целое число) и записывать соответственно в переменные <i>Chameleon1</i> и <i>Chameleon2</i> . Вывести значения и типы переменных на экран |
| 1.2 | Преобразовать переменную <i>Chameleon2</i> к строковому типу с использованием функции <i>str()</i> . Умножить полученную строку на число даты рождения пользователя и вывести результат |
| 1.3 | Сложить переменные <i>Chameleon1</i> и <i>Chameleon2</i> , результат записать в <i>Chameleon3</i> . Вывести результат суммирования и тип переменной |

| | |
|------|--|
| 1.4 | Преобразовать переменную <i>Chameleon3</i> к списку с использованием функции <code>list()</code> . Вывести результат и тип переменной на экран |
| 1.5 | Напечатать начальный и конечный элементы списка <i>Chameleon3</i> . Первое значение списка <i>Chameleon3</i> заменить возрастом пользователя |
| 1.6 | Прибавить к списку <i>Chameleon3</i> список, вводимый пользователем с клавиатуры. Результат вывести на экран, а затем умножить на 2 и снова вывести на экран |
| 1.7 | Преобразовать переменную <i>Chameleon3</i> к кортежу с использованием функции <code>tuple()</code> . Вывести результат и тип переменной на экран |
| 1.8 | Попытаться изменить любое значение кортежа <i>Chameleon3</i> |
| 1.9 | Продемонстрировать операцию сложения кортежей и умножения кортежа на число |
| 1.10 | Преобразовать переменную <i>Chameleon3</i> к множеству с использованием функции <code>set()</code> . Вывести результат и тип переменной на экран. Сделать вывод о назначении множеств |
| 1.11 | Создать словарь <i>My_Dictionary</i> из пяти пар, осуществляющий перевод слов с русского на английский. Продемонстрировать обращение к элементам словаря по ключу. Вывести словарь несколько раз на экран. Сделать вывод |
| 1.12 | Сделать вывод о простых типах данных языка <i>Python</i> , универсальности команд и о динамической типизации |

2. Разработать интерактивную программу «*What is My Age in Seconds*» («Каков мой возраст в секундах»), которая на входе принимает дату рождения пользователя, рассчитывает количество прожитых пользователем секунд и выводит результат на экран монитора.

Дополнительное задание

Согласно официальному сайту Европейского Центрального Банка в настоящий момент наши западные соседи используют в обращении 7 номиналов банкнот евро – 5, 10, 20, 50, 100, 200 и 500 евро. Монетный ряд (см. рис. 3.4) включает 8 монет достоинством 1, 2, 5, 10, 20, 50 евроцентов, 1 и 2 евро (1 евро = 100 евроцентов).

Вариант 1. Необходимо написать программу для специального банкомата (АТМ). Банкомат должен оптимальным способом выдавать любую введенную пользователем сумму: вначале выдаются крупные банкноты, потом меньше и так вплоть до одного евроцента.

К примеру, пользователь запрашивает в банкомате сумму в 587 евро и 99 центов. Банкомат ему должен выдать следующее: банкноты номиналом 500, 50, 20, 10 и 5 евро, затем монеты: 50, 20×2 , 5, 2×2 и 1 евроцент.

Вариант 2. Необходимо написать приложение для разменного автомата. Работа автомата заключается в следующем: автомат принимает у пользователя купюру и выдает ему купюры меньшего номинала. Аналогично осуществляется размен монет.

Замечания.

1. Студенты, имеющие нечетный номер варианта выполняют первый вариант задания, остальные – второй.
2. Допускается использовать любую другую валюту.

Требования к выполнению

1. Программа должна быть обязательно снабжена комментариями на английском языке, в которых необходимо указать краткое предназначение программы, номер лабораторной работы и ее название, версию программы, Ф.И.О. разработчика, дату разработки.

2. Каждая программа должна быть снабжена дружелюбным и интуитивно понятным интерфейсом (да-да, пусть даже пока в консольном варианте).



Перед началом работы необходимо всегда думать о нуждах конечных пользователей, только так можно создать по-настоящему качественный продукт!

Контрольные вопросы

1. Опишите архитектуру и основные элементы компьютера. Какой элемент является центральным при построении любой вычислительной системы?
2. Какие типы памяти доступны при разработке программ?
3. Что такое машинный код?
4. В чем отличия языка высокого уровня от языка низкого уровня?
5. Какой язык понимает и обрабатывает центральный процессор (*Central Process Unit, CPU*)?

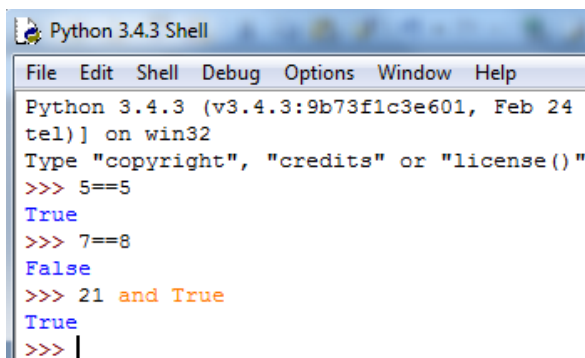
6. Что такое транслятор и что он делает?
7. Что общего между компилятором и интерпретатором и чем они отличаются?
8. Что такое переменная и зачем она нужна в программе?
9. Что такое «соглашение по присваиванию имен» ?
10. Что такое константа?
11. Что такое *hardcode* («тяжелый код»)?
12. Описать стандартные простые типы данных в *Python*.
13. Что измеряет диапазон типа данных?
14. Что такое выражение?
15. Что такое инициализация?
16. Можно ли обратиться к переменной, если инициализировать ее чуть позже?
17. Что произойдет, если присвоить строковой переменной целочисленное значение (или наоборот)?
18. Какой из арифметических операторов не может иметь дробного операнда?
19. Какой из арифметических операторов не может иметь в качестве второго операнда значение ноль?
20. Описать все группы операций, доступные в языке *Python*.

4. УСЛОВНЫЕ КОНСТРУКЦИИ *PYTHON*

4.1. Базовый синтаксис языка *Python*: условные конструкции

Логические выражения

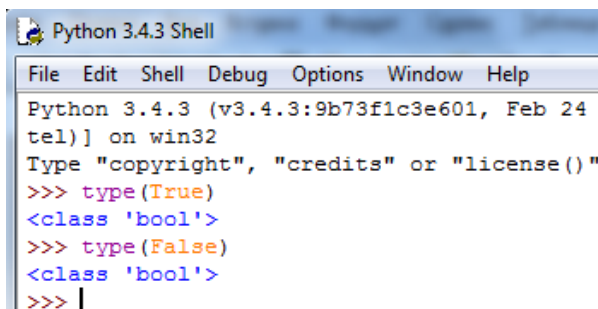
Логическими (boolean expression) называются выражения, которые могут принимать одно из двух значений – истина или ложь. Операнды логических операторов должны быть логическими выражениями. В *Python* любое ненулевое число интерпретируется им как «истинное» (рис. 4.1).



```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24
tel)] on win32
Type "copyright", "credits" or "license()"
>>> 5==5
True
>>> 7==8
False
>>> 21 and True
True
>>> |
```

Рис. 4.1. Пример логических выражений

True и *False* – специальные значения, которые принадлежат к типу *bool*, они не являются строками (рис. 4.2).



```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24
tel)] on win32
Type "copyright", "credits" or "license()"
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
>>> |
```

Рис. 4.2. *True* и *False*

Условное исполнение

Условные инструкции (*conditional statements*) позволяют изменять ход выполнения программы в зависимости от условий.

На рис. 4.3 представлена блок-схема и код для проверки положительности числа x с выводом соответствующего сообщения.

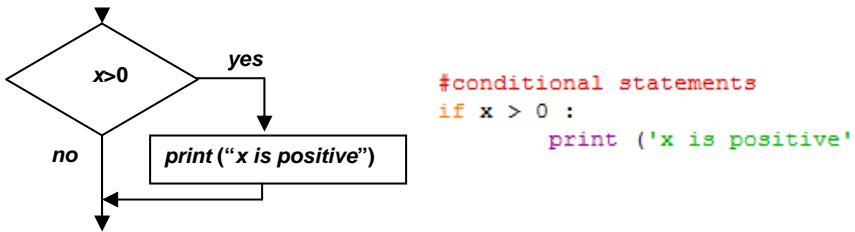


Рис. 4.3. Блок-схема условной конструкции и код на языке *Python*

Логическое выражение после инструкции `if` называется *условием*, далее следует *символ двоеточия* (`:`) и *строка (строки) с отступом*. Если логическое условие истинно, то управление получает выражение, записанное с отступами, иначе – выражение пропускается.

Не существует ограничения на число инструкций, которые могут встречаться в теле `if`, но хотя бы одна инструкция там должна быть. Иногда полезно иметь тело `if` без инструкций (обычно оставляют место для кода, который еще не написан). В этом случае можно воспользоваться инструкцией `pass`, которая ничего не делает (рис. 4.4).

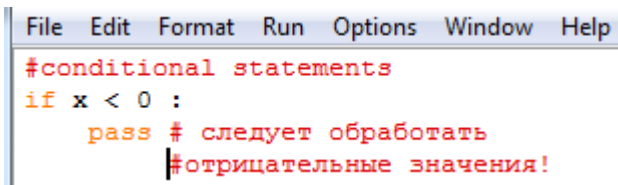


Рис. 4.4. Инструкция *pass*

Альтернативное исполнение

Альтернативное исполнение (*alternative execution*) инструкции `if` предполагает два направления выполнения, и условие определяет, ка-

кое из них выполнится. Пример (рис. 4.5) иллюстрирует проверку четности/нечетности числа x с выводом соответствующего сообщения.

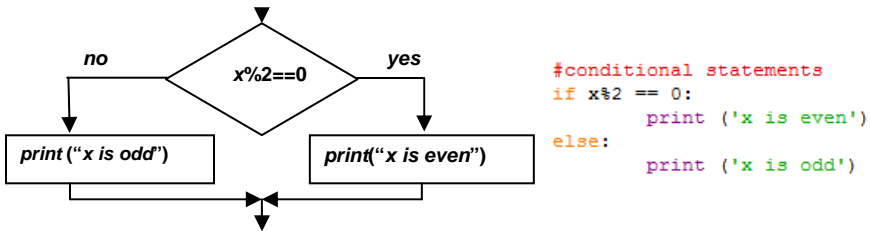


Рис. 4.5. Инструкция `if`

Альтернативное исполнение предполагает выполнение одного из вариантов, так как условие может быть либо истинным, либо ложным. Варианты называются *ветвями* (*branches*).

Последовательность условий

Когда имеется больше двух вариантов выполнения, то необходимо больше двух ветвей. В этом случае, можно воспользоваться *сцепленными условиями* (*chained conditional*). На рис. 4.6 приведен алгоритм сравнения двух чисел x и y .

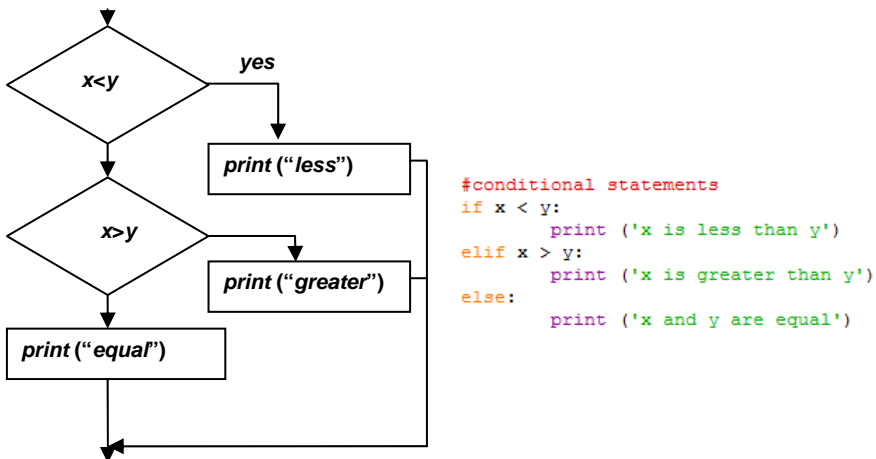


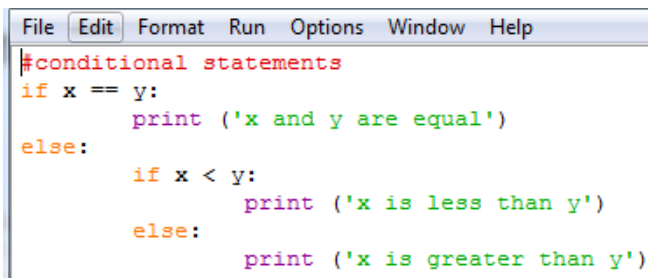
Рис. 4.6. Сцепленные условия

Инструкция `elif` – аббревиатура от «else if». Оператор `else` должен быть в конце инструкции, но может и отсутствовать.

Каждое условие проверяется в порядке расположения. Если первое условие ложно, то проверяется следующее и так далее. Если одно из условий истинно, то выполняется соответствующая ветка, и инструкция завершается.

Вложенные условия

Условия могут быть вложенными. Пример *трихотомии*, иллюстрирующий сравнение чисел x и y , представлен рис. 4.7.

A screenshot of a code editor window with a menu bar (File, Edit, Format, Run, Options, Window, Help) and a text area containing Python code. The code implements a trichotomy function that compares two variables, x and y. It uses nested if-else statements to print messages based on whether x is equal to, less than, or greater than y. The code is color-coded: comments are red, keywords are orange, strings are green, and print statements are purple.

```
#conditional statements
if x == y:
    print ('x and y are equal')
else:
    if x < y:
        print ('x is less than y')
    else:
        print ('x is greater than y')
```

Рис. 4.7. Код трихотомии

Хотя отступ инструкций делает структуру более очевидной, *вложенные условия* (*nested conditionals*) усложняют чтение кода. Их следует по возможности избегать.

Перехват исключений с помощью *try* и *except*

Структура условного выполнения «`try/except`» обрабатывает типы ожидаемых и неожиданных ошибок. Идея `try` и `except` заключается в следующем. Пусть некоторая последовательность инструкций может иметь ошибки. Нужно добавить дополнительные инструкции (блок `except`), которые выполняются в случае возникновения ошибки, и игнорируются, если ошибка не произошла.

Общий вид конструкции «`try/except`»:

```
try:
    instruction 1
except:
    instruction 2
```

Выполнение начинается с последовательности инструкций в блоке `try`. Если все выполняется без ошибок, блок `except` пропускается. Если произошло исключение в блоке `try`, то блок `try` покидается и выполняет последовательность инструкций внутри блока `except`. Обработка исключения с помощью инструкции `try` называется *перехватом* (*catching*) исключения.

Пример кода и результат программы ввода с учетом некорректного ввода представлен на рис. 4.8.

The image shows two windows. The left window is a text editor titled `*try_ex.py - D:/try_ex.py (3.4.3)*` containing the following code:

```
#catching
try:
    x=int(input('Введите целое число\n'))
except:
    print('Некорректный ввод данных')
```

The right window is a terminal titled `Python 3.4.3 Shell` showing the execution of the script:

```
Python 3.4.3 (v3.4.3:9b73f1c3e601,
tel) on win32
Type "copyright", "credits" or "lic
>>> =====
>>>
Введите целое число
lololo
Некорректный ввод данных
>>> |
```

Рис. 4.8. Учет некорректного ввода целого числа с клавиатуры

Случайные числа

Программы, генерирующие одни и те же выходные значения для одинаковых входных значений, называются *детерминированными*. Для многих приложений, например игр, характерна непредсказуемость в поведении. Для создания недетерминированных программ можно использовать *алгоритмы генерации псевдослучайных чисел*.

Модуль `random` предоставляет функции, которые генерируют псевдослучайные числа (далее «случайные числа»):

- `random()` возвращает случайное число с плавающей точкой в интервале от 0.0 до 1.0 (включая 0.0 и не включая 1.0);
- `randint(low, high)` возвращает целочисленное значение в интервале от `low` до `high` включительно;
- `randrange(start, stop, step)` возвращает число от `start` включительно до `stop` не включительно с шагом `step`;
- для выбора случайного элемента из последовательности, можно воспользоваться функцией `choice()`.

Модуль `random` также включает функции, которые генерируют значения, принадлежащие показательному закону распределения, распределению Гаусса, Гамма и другие.

4.2. Лабораторная работа 4

«Условные конструкции *Python*»

Цель работы: изучить синтаксис условных конструкций языка *Python*, продемонстрировать возможности конструкций ветвления на примере разработке интерактивных приложений.

Основное задание

Разработать интерактивную программу «*Quadratic Equation*» («Квадратное уравнение») для решения квадратных уравнений вида: $ax^2 + bx + c = 0$. Программа должна запрашивать соответствующие параметры a , b и c , проверять параметры и выдавать результат.

Индивидуальное задание

В соответствии с заданием своего варианта выполнить задания из прил. А.

Дополнительное задание

1. Написать программу, которая бы эмулировала игру «*Dice*» (игра в кости). Суть игры заключается в броске двух шестигранных кубиков (костей) и подсчета общей суммы очков, которые выпали на первой и второй кости. Для генерирования случайного значения на костях воспользоваться функциями из стандартного модуля `random`.

2. Написать программу – симулятор пирожков с «сюрпризом». Программа должна выводить пирожок и один из пяти (можно больше) различных «сюрпризов», который бы выбирался случайно.

3. Написать программу «*Mood Sensor*», которая определяет настроение пользователя в текущий момент времени. Приложение генерирует случайное число, в зависимости от которого выводится одно из псевдографических «лиц», отображающее настроение.

Требования к выполнению

1. Программа должна обязательно быть снабжена комментариями на английском языке, в которых необходимо указать краткое предназначение программы, номер лабораторной работы и ее название, версию программы, Ф.И.О. разработчика и дату разработки.

2. Каждая программа должна быть снабжена дружелюбным и интуитивно понятным интерфейсом (да-да, пусть даже пока в консольном варианте).



Теперь Вы знаете условные конструкции языка *Python*!

Контрольные вопросы

1. Перечислите основные управляющие конструкции.
2. Опишите основные элементы блок-схем.
3. Как в языке *Python* реализуется механизм истинности-ложности? Может ли значение быть условием?
4. Опишите синтаксис простой условной конструкции `if`. Представьте примерную блок-схему конструкции.
5. С помощью каких операторов можно комбинировать в одной условной конструкции `if` несколько условий? Какой механизм оптимизации применяет интерпретатор *Python* для эффективного вычисления результата комбинированных условных выражений?
6. Опишите синтаксис условной конструкции `if-else`. Представьте примерную блок-схему конструкции.
7. Опишите синтаксис условной конструкции `elif`. Представьте примерную блок-схему конструкции.
8. Чем использование `elif` будет отличаться от использования вложенных условных конструкций `if-else`?
9. Как сгенерировать случайную последовательность чисел с использованием функций `randint()` и `randrange()`? Чем они отличаются? Какие еще есть полезные функции в модуле *random*?

5. ЦИКЛИЧЕСКИЕ КОНСТРУКЦИИ. ИТЕРАЦИОННЫЕ АЛГОРИТМЫ

5.1. Цикл *while*

Инструкция *while*

Одной из форм организации итераций в *Python* является инструкция *while*, общий синтаксис которой имеет вид:

```
While condition:
instruction 1
...
instruction n
```

Поток выполнения для инструкции *while* имеет вид:

- 1) проверка условия (*condition*) (*True* или *False*);
- 2) если условие ложно, происходит выход из инструкции *while*, и выполнение продолжается со следующей инструкции;

3) если условие истинно, выполняется тело цикла (*instruction 1, ... , instruction n*) и происходит возврат к шагу 1. Такой поток называется циклом (*loop*), так как шаг 3 возвращает к началу алгоритма. Выполнение тела цикла называется *итерацией* (*iteration*). Итерации выполняются, пока условие истинно.

Код и результат программы, производящей обратный отсчет от 5 до 1 с выводом сообщения «*Blastoff!*», представлен на рис. 5.1. Для цикла выполнено 5 итераций.

| | |
|---|---|
| <pre>File Edit Format Run # Blastoff!! #While_loop n = 5 while n > 0: print (n) n-=1 print ('Blastoff!')</pre> | <pre>File Edit Shell Debug Options Window Help Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 tel)] on win32 Type "copyright", "credits" or "license()" >>> ===== RESTA >>> 5 4 3 2 1 Blastoff! >>> </pre> |
|---|---|

Рис. 5.1. Код и результат программы, производящей обратный отсчет от 5 до 1 с выводом сообщения «*Blastoff!*»

Тело цикла должно изменять одну или более переменных, что в конечном итоге должно привести к ложности условия и завершению цикла. Переменные, которые изменяются каждый раз при выполнении цикла, и контролируют завершение цикла, называются *итерационными переменными* (*iteration variable*). Если итерационная переменная в цикле отсутствует, то такой цикл будет *бесконечным* (*infinite loop*).

Замечание. Обновление переменной путем прибавления к ней 1 называется *инкрементом* (*increment*), вычитание 1 называется *декрементом* (*decrement*).

Бесконечные циклы и *break*

Бесконечный цикл не содержит итерационную переменную, которая указывает на количество выполнений в цикле. Он используется в тех случаях, когда только при выполнении очередной итерации становится понятно, надо ли завершить цикл.

Инструкция `break` используется, когда нужно выйти из бесконечного цикла при наступлении заданного условия:

```
While True:
    instruction 1
    ...
    break
    ...
    instruction n
```

Если не добавить `break` в тело цикла, то произойдет «защелкивание» и цикл будет выполняться бесконечно.

Код и результат программы, которая бесконечно информирует пользователя, чему равна скорость света в вакууме, представлен на рис. 5.2–5.4.

A screenshot of a Python IDE window. The title bar shows a menu with 'File', 'Edit', 'Format', 'Run', 'Options', 'Window', and 'Help'. The code editor contains the following Python code:

```
while True:
    print ('Скорость света в вакууме равна 299792458 м/с!')
```

Рис. 5.2. Вывод бесконечного сообщения

```
File Edit Format Run Options Window Help
while True:
    x=float(input('Чему равна скорость света в вакууме (м/с)?\n'))
    if x!=299792458:
        print ('А вот и нет! Давайте попробуем еще раз!')
    else:
        print ('Отличные знания физики! (или умение искать в google:))')
        break
print('Поздравляем с правильным ответом!')
```

Рис. 5.3. Вывод бесконечного сообщения с условными операторами

```
...
Чему равна скорость света в вакууме (м/с)?
56789777
А вот и нет! Давайте попробуем еще раз!
Чему равна скорость света в вакууме (м/с)?
59877755778
А вот и нет! Давайте попробуем еще раз!
Чему равна скорость света в вакууме (м/с)?
299792458
Отличные знания физики! (или умение искать в google:))
Поздравляем с правильным ответом
>>>
```

Рис. 5.4. Результат работы программы о скорости света в вакууме

Завершение итерации с помощью continue

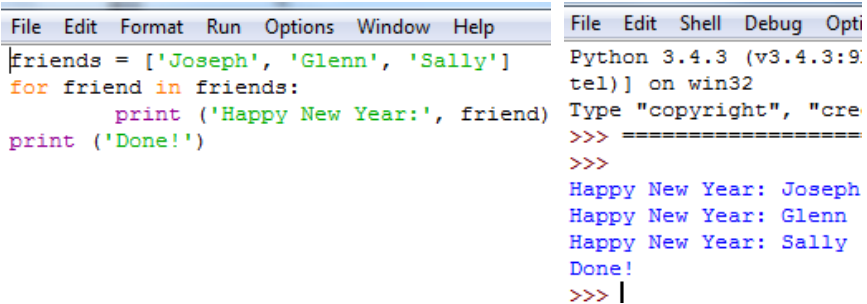
Инструкция `continue` пропускает текущую итерацию в цикле и переходит к следующей без завершения тела цикла. На рис. 5.5 представлен цикл, который копирует данные, поступающие на вход до тех пор, пока не будет введено «done». Если строка начинается с символа «#», то она не выводится на печать.

```
while True:
    line = input('Введите строку\n')
    if line[0] == '#':
        continue
    if line == 'done':
        break
    print (line)
print ('Done!')
```

Рис. 5.5. Пример использования инструкции `continue`

5.2. Цикл *for*

Цикл `for` проходит через известное множество элементов столько раз, сколько элементов содержится во множестве. Пример кода и результата программы представлен на рис. 5.6.

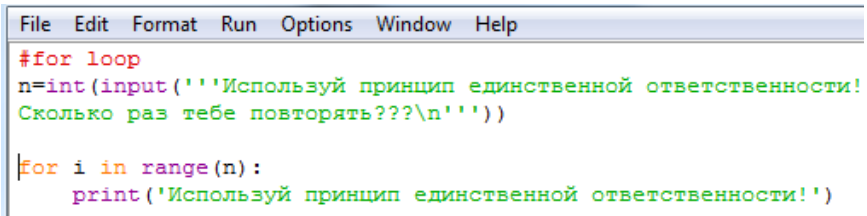


```
File Edit Format Run Options Window Help | File Edit Shell Debug Opti
friends = ['Joseph', 'Glenn', 'Sally']
for friend in friends:
    print ('Happy New Year:', friend)
print ('Done!')
```

```
Python 3.4.3 (v3.4.3:9
tel)] on win32
Type "copyright", "cre
>>> =====
>>>
Happy New Year: Joseph
Happy New Year: Glenn
Happy New Year: Sally
Done!
>>> |
```

Рис. 5.6. Пример использования цикла *for*

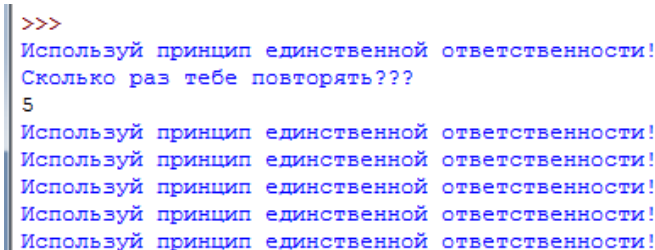
Конструкция `in range(n)` используется для цикла прохождения определенного числа шагов (n) (рис. 5.7, 5.8). Заметим, что итерационная переменная меняется от 0 до $n - 1$.



```
File Edit Format Run Options Window Help
#for loop
n=int(input('Используй принцип единственной ответственности!
Сколько раз тебе повторять???\n'))

for i in range(n):
    print('Используй принцип единственной ответственности!')
```

Рис. 5.7. Пример использования конструкции `in range(n)` в цикле `for`



```
>>>
Используй принцип единственной ответственности!
Сколько раз тебе повторять???
5
Используй принцип единственной ответственности!
Используй принцип единственной ответственности!
Используй принцип единственной ответственности!
Используй принцип единственной ответственности!
Используй принцип единственной ответственности!
```

Рис. 5.8. Результат исполнения конструкции `in range(n)` в цикле `for`

5.3. Лабораторная работа 5

Часть 1. «Циклические конструкции. Итерационные алгоритмы. Цикл *while*»

Цель работы: изучить синтаксис циклической конструкции `while` языка Python для программирования итерационных алгоритмов, продемонстрировать возможности конструкции `while` на примере разработке интерактивных приложений.

Основное задание

Разработать интерактивную программу «*Try to Guess the Number*» (Попробуй угадать число), которая эмулирует игру на отгадывание числа. Суть игры сводится к следующему: компьютер генерирует случайное число из диапазона, к примеру, от 1 до 100, а пользователь пытается отгадать число. При каждой попытке компьютер «подсказывает» игроку, как соотносится вариант игрока с загаданным компьютером числом: загаданное число больше или меньше указанного. Как только игрок отгадывает число, компьютер должен «поздравить» его с выводом на экран угаданного числа и количества затраченных игроком попыток. Далее компьютер может «предложить» повторно сыграть в игру или выйти.

Для универсальности можно добавить возможность выбора диапазона генерирования компьютером случайных чисел, а также задание ограничения на количество попыток. В случае, если игрок не укладывается в заданное количество попыток, программа должна выводить надпись «*Game Over*» из лабораторной работы 2.

Индивидуальное задание

В соответствии с заданием своего варианта выполнить задания из прил. В

Дополнительное задание

1. Модернизировать программы, которые были разработаны в предыдущих лабораторных работах: «*ATM*» (банкомат) и «*Coin changer*» (разменный аппарат), «*Dice*» (игра в кости), симулятор пирожков с сюрпризом и «*Mood Sensor*» (датчик настроения). Их нужно переписать таким образом, чтобы у пользователя была возможность повторного выполнения программы без выхода из нее.

2. Написать программу «*Heads or Tails?*» (Орел или решка?), которая бы «подбрасывала» условно монету, к примеру, 1000 раз и сообщала, сколько раз выпал орел, а сколько – решка.

3. Переработать программу из основного задания таким образом, чтобы пользователь загадывал число, а компьютер, используя оптимальный алгоритм, число отгадывал.

Требования к выполнению

1. Программа должна обязательно быть снабжена комментариями на английском языке, в которых необходимо указать краткое предназначение программы, номер лабораторной работы и ее название, версию программы, Ф.И.О. разработчика и дату разработки.

2. В программах, где это необходимо, предусмотреть возможность ее повторного выполнения, а также защиты от ввода некорректных данных (так называемую «защиту от дурака»).

3. Каждая программа должна быть снабжена дружелюбным и интуитивно понятным интерфейсом.

Контрольные вопросы

1. Для чего используются циклы? Что такое итерация?
2. Какие разновидности циклов существуют?
3. Описать *Python*-синтаксис цикла с предусловием `while`.
4. Чем является выражение после ключевого слова `while` – инициализацией, условием или обновлением?
5. Какова роль оператора `break` в теле цикла?
6. Какова роль оператора `continue` в теле цикла?
7. Какова роль оператора `pass` в теле цикла?
8. Может ли выражение после ключевого слова `while` содержать истинное значение или значение других типов данных?
9. Что такое бесконечный цикл? Когда он применяется? Привести пример кода организации диалога на тему завершения программы, либо повторного выполнения программы.
10. Если необходимо использовать вложенные циклы `while` для вывода элементов прямоугольной матрицы в виде строк и столбцов, какой из циклов будет печатать строки: внутренний или внешний?

5.4. Лабораторная работа 5

Часть 2. «Циклические конструкции. Итерационные алгоритмы. Цикл *for*»

Цель работы: изучить синтаксис циклической конструкции `for` языка *Python* для программирования итерационных алгоритмов, продемонстрировать возможности циклических конструкций *while* и *for* на примере разработке интерактивных приложений.

Основное задание

Разработать программы для решения следующих задач, которые должен уметь реализовать каждый программист (*it's easy...*):

1. Найти сумму цифр и количество цифр заданного натурального числа.
2. Возвести число в натуральную степень n .
3. Найти количество различных цифр у заданного натурального числа.
4. Найти наибольшую цифру натурального числа.
5. Задано натуральное число. Проверить, является ли заданное натуральное число палиндромом.
6. Определить является ли заданное натуральное число простым.
7. Найти все простые делители заданного натурального числа.
8. Найти НОД и НОК двух натуральных чисел.
9. Заданы три целых числа, которые задают некоторую дату. Определить дату следующего дня.
10. Запрограммировать последовательность чисел Фибоначчи (пользователь вводит порядковый номер элемента последовательности Фибоначчи, а программа выводит на экран значение).

Дополнительное задание

1. Совершенным называется число, равное сумме всех своих делителей, не равных самому числу, учитывая и 1. Проверить является ли заданное число совершенным.
2. Задано число, содержащее от двух и более цифр. Между каждой парой соседних цифр, вставить заданное число. Например, число 7: $243 \rightarrow 27473$.

3. Задано натуральное число N . Каждое вхождение наибольшей цифры, использованной в записи числа N , продублировать. Например, 349291 \rightarrow 34992991.

Требования к выполнению

1. Программа должна обязательно быть снабжена комментариями на английском языке, в которых необходимо указать краткое предназначение программы, номер лабораторной работы и ее название, версию программы, Ф.И.О. разработчика, номер группы и дату разработки.

2. В программах, где это необходимо, предусмотреть возможность ее повторного выполнения и защиту от ввода некорректных данных (предусмотреть так называемую «защиту от дурака»).

3. Для реализации всех программ использовать только числовые типы данных *Python* (*int*, *long*, *float*, *bool*) и строки (*str*).

4. Каждая программа должна быть снабжена дружелюбным и интуитивно понятным интерфейсом.

Контрольные вопросы

1. Опишите *Python*-синтаксис и принцип работы цикла *for*.

2. В чем особенность использования циклической конструкции *for* в языке *Python* по сравнению с использованием в других языках программирования, к примеру, в *Pascal* или *C/C++*?

3. Для чего используется стандартная функция *range* (. .) ?

4. Опишите наиболее распространенные случаи использования функции *range* (. .).

5. Опишите сочетание функции *range* (. .) и цикла *for*.

6. Обязательно ли использовать переменную цикла *for* внутри самого цикла?

7. Какую дополнительную конструкцию (ветку) могут иметь циклы в языке *Python*? В каких случаях тело данной конструкции будет выполняться, а в каких – нет?

6. ФУНКЦИИ. РЕКУРСИВНЫЕ АЛГОРИТМЫ

6.1. Функции в языке *Python*. Рекурсивные алгоритмы. Модули

Функции. Вызов функции

В контексте программирования *функцией* (*function*) называется хранимая последовательность инструкций, предназначенная для решения определенной задачи. Основными параметрами функции являются:

- имя функции;
- тело функции;
- передаваемые параметры (аргументы);
- возвращаемые параметры (результат).

В *Python* предоставляется возможность создавать свои собственные (пользовательские) функции, а также работать со встроенными стандартными функциями (функции ввода/вывода данных, функции преобразования типов, математические функции модуля *math*, функции генерации псевдослучайных функций модуля *random*).

Правила наименования функций такие же, как для переменных, например, нельзя использовать зарезервированные слова в качестве имен функций. Имена функций и переменных не должны совпадать.

Первая строка определения функции называется *заголовком* (*header*), оставшаяся часть – *телом* (*body*) функции. Заголовок заканчивается двоеточием, тело функции имеет отступ. Тело функции может содержать любое количество инструкций. Инструкции внутри функции не получают управления, пока функция не будет вызвана. Для возврата результата функции, используется инструкция *return*. Вызвать функцию (*function call*) можно, обратившись к ней по имени.

Например, на рис. 6.1 представлена функция *addtwo()*, имеющая два аргумента, и обращение к ней. Указанная функция складывает два числа, и возвращает результат. Для вызова этой функции ей нужно передать два параметра. Возвращаемый параметр функции – переменная *added*, в которую записывается сумма аргументов.

```
File Edit Format Run
#addition function
def addtwo(a, b):
    added = a + b
    return added

sum=addtwo(5, 3)#function call
```

```
File Edit Format Run Options Windo
#addition function
def addtwo(a, b):
    added = a + b
    return added

sum=addtwo(5, 3)#function call
```

Рис. 6.1. Функция addtwo и ее вызов

Пустые скобки после имени функции указывают на то, что функция не требует аргументов, например, функция `random()`. Если попытаться присвоить результат выполнения такой функции переменной, вернется специальное значение, называемое `None`.

Существуют функции, не возвращающие результат, например, функция `print()` (рис. 6.2).

```
File Edit Shell Debug C
Python 3.4.3 (v3.4.3
tel)] on win32
Type "copyright", "c:
>>> =====
>>>
>>> type(print())

<class 'NoneType'>
>>> |
```

Рис. 6.2. Функция `print()`

На рис. 6.3 приведен пример функции `add_mult_two(a, b)`, возвращающей несколько значений.

```
File Edit Format Run Options Window Help
#addition & multiplication function
def add_mult_two(a, b):
    added = a + b
    mult=a*b
    return added, mult

sum, prod =add_mult_two(5, 3)#function call
```

Рис. 6.3. Функция `add_mult_two(a, b)`

Поток исполнения

Для того чтобы убедиться, что функция определяется до ее первого использования, нужно знать порядок, в котором выполняются инструкции, он называется *поток исполнения (flow of execution)*. Исполнение обычно начинается с первой инструкции программы.

Вызов функции подобен обходному пути в потоке исполнения. Вместо того чтобы перейти к следующей инструкции, поток переходит в тело функции, выполняет все инструкции внутри тела, и затем возвращается обратно в то место, которое он покинул в момент вызова функции. При этом в процессе вызова одной функции, программа может выполнять инструкции из другой.

Принцип единственной ответственности

Программу рекомендуется разбивать на функции, поскольку:

- создание новой функции предоставляет возможность присвоить имя группе инструкций, что позволит упростить чтение, понимание и отладку программы;
- функции позволяют сократить код программы, благодаря ликвидации повторяющихся участков кода;
- разбиение длинной программы на функции позволяет одновременно отлаживать отдельные части, а затем собрать их в единое целое;
- хорошо спроектированная функция может использоваться в других программах.

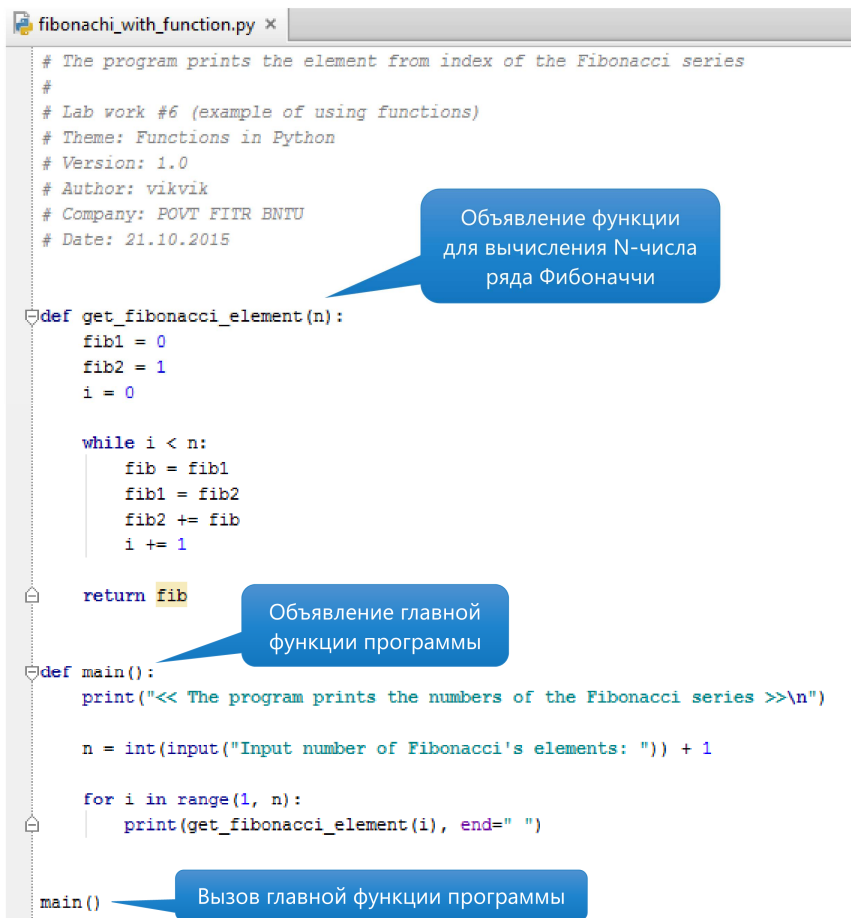
Принцип единственной ответственности (The Single Responsibility Principle) декларирует разграничение ответственности между функциями: за решение одной конкретной задачи должна отвечать одна функция.

Использование принципа единственной ответственности при работе с функциями является хорошим стилем программирования.

Более детально принцип рассматривается при изучении основ объектно-ориентированного программирования.

6.2. Пример выполнения задания для вычисления элементов ряда Фибоначчи

Пусть необходимо реализовать функцию, которая вычисляет первые N -элементов ряда Фибоначчи, а также написать программу для ее тестирования. Реализация представлена на рис. 6.4.



```
fibonachi_with_function.py ×  
  
# The program prints the element from index of the Fibonacci series  
#  
# Lab work #6 (example of using functions)  
# Theme: Functions in Python  
# Version: 1.0  
# Author: vikvik  
# Company: POVT FITR BNTU  
# Date: 21.10.2015  
  
def get_fibonacci_element(n):  
    fib1 = 0  
    fib2 = 1  
    i = 0  
  
    while i < n:  
        fib = fib1  
        fib1 = fib2  
        fib2 += fib  
        i += 1  
  
    return fib  
  
def main():  
    print("<< The program prints the numbers of the Fibonacci series >>\n")  
  
    n = int(input("Input number of Fibonacci's elements: ")) + 1  
  
    for i in range(1, n):  
        print(get_fibonacci_element(i), end=" ")  
  
main()
```

Объявление функции для вычисления N -числа ряда Фибоначчи

Объявление главной функции программы

Вызов главной функции программы

Рис. 6.4. Код программы решения задания

6.3. Пример выполнения задания с использованием рекурсивного подхода

Необходимо разработать программу, которая вычисляет факториал числа N с использованием двух способов: итеративного и рекурсивного. Этапы реализации представлены на рис. 6.5, 6.6.

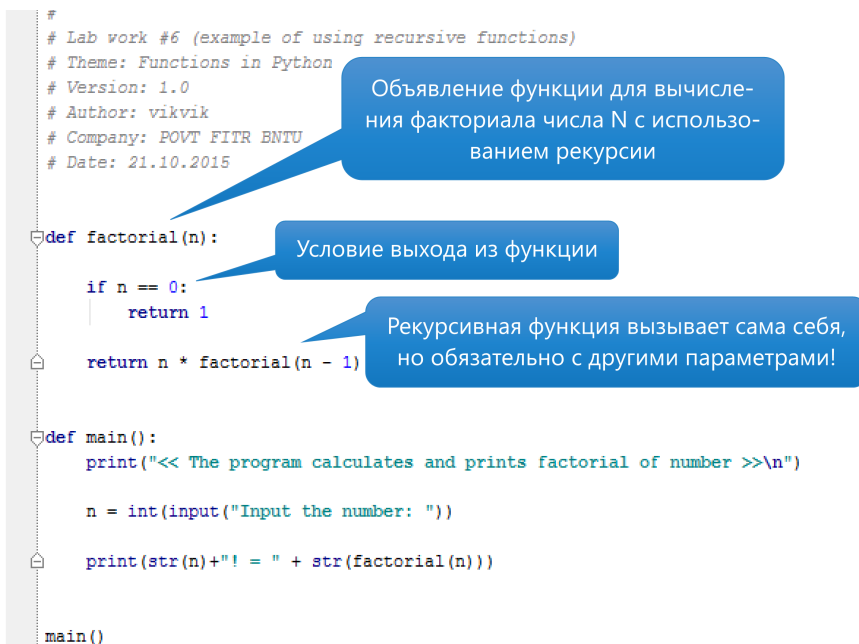


Рис. 6.5. Вычисление факториала числа N рекурсивным методом

```

def factorial(n):
    p = 1
    for i in range(1, n + 1):
        p *= i
    return p

def main():
    print("<< The program calculates and prints factorial of number >>\n")
    n = int(input("Input the number: "))
    print(str(n)+"! = " + str(factorial(n)))

main()

```

Рис. 6.6. Вычисление факториала итерационным методом

6.4. Лабораторная работа 6 «Функции. Рекурсивные алгоритмы»

Цель работы: познакомиться с наиболее востребованными стандартными функциями *Python*, изучить синтаксис объявления и использования пользовательских функций; познакомиться с рекурсивными алгоритмами; научиться проектировать и разбивать большую программу на мелкие фрагменты (функции); закрепить знания на примере разработки интерактивных приложений.

Основное задание

1. Рекурсивно описать функцию $f(x, n)$, вычисляющую $x^n/n!$ при любом действительном x и любом неотрицательном целом n .
2. Рекурсивно описать функцию $pow(x, n)$, вычисляющую x^n для любого действительного $x \neq 0$ и любого целого n .
3. Реализовать функцию, которая вычисляет N -й элемент ряда Фибоначчи с использованием рекурсивного алгоритма. На базе данной функции разработать программу, которая должна предлагать пользователю следующие возможности:
 - 1) вывод конкретного элемента последовательности;
 - 2) вывод всех элементов до указанного пользователем элемента;

3) вывод части последовательности, значение последнего элемента которой не превосходит введенного пользователем значения.

Индивидуальное задание

Модернизировать программы и оптимизировать алгоритмы решения основных и дополнительных заданий лабораторной работы 5 (часть 2), используя функции (возможно рекурсию).

Дополнительное задание

1. Найти все натуральные числа, не превосходящие N , сумма цифр каждого из которых в некоторой степени дает это число (например: $7^4 = 2401$, $8^3 = 512$, $9^2 = 81$ и т. д.).

2. Напечатать N автоморфных чисел (числа, совпадающие с младшими цифрами своего квадрата, например: 25 и 625).

3. Найти N троек чисел Пифагора (натуральные числа a , b и c называются числами Пифагора, если выполняется условие $a^2 + b^2 = c^2$).

4. Описать рекурсивную функцию *equals* (N , S) (где N и S – неотрицательные целые числа), которая проверяет, совпадает ли сумма цифр в десятичной записи числа N со значением S (например: *equals* (1234567, 28) = *True*, *equals* (10, 7) = *False*).

5. Рекурсивно описать функцию *divs* (N) для подсчета количества всех делителей заданного целого числа N ($N > 1$), без учета делителей 1 и N (например: *divs* (7) = 0, *divs* (10) = 2, *divs* (16) = 3 и так далее).

6. Задача о Ханойской башне (*Tower of Hanoi*). Имеется три стержня А, В, С, на один из которых нанизаны N колец, причем кольца отличаются размером и лежат меньшее на большем. Требуется перенести пирамиду из N колец с одного стержня на другой за наименьшее число ходов. За один раз разрешается переносить только одно кольцо, причем нельзя класть большее кольцо на меньшее.

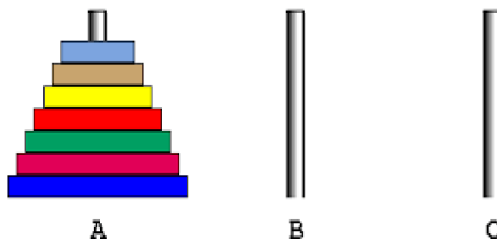


Рис. 6.7. Задача о Ханойской башне

Например, для функции `Hanoi (3,'A','B','C')`, где 7 – число колец, А – стержень-источник, В – стержень-приемник, С – имя временного стержня, должен получиться ответ, как на рис. 6.8:

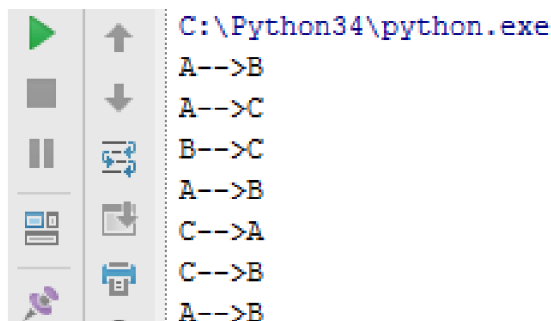


Рис. 6.8. Решение задачи о Ханойской башне



Для тех, кто хочет закрепить свои знания и навыки для решения задач с помощью рекурсий, могут быть полезны ресурсы:

<http://server.179.ru/tasks/training/recursion.html>,

<https://ru.wikibooks.org/wiki/>

Требования к выполнению

1. Программа должна обязательно быть снабжена комментариями на английском языке, в которых необходимо указать краткое предназначение программы, номер лабораторной работы и ее название, версию программы, Ф.И.О. разработчика и дату разработки.

2. В программах по необходимости предусмотреть возможность ее повторного выполнения, а также защиты от ввода некорректных пользовательских данных (предусмотреть «защиту от дурака»). Для этих вещей рекомендуется разработать отдельные функции;

3. При разработке программ рекомендуется придерживаться принципа единственной ответственности, то есть функции должны проектироваться и реализовываться таким образом, чтобы они были менее завязаны с другими функциями при своей работе.

4. Каждая программа должна быть снабжена дружелюбным и интуитивно понятным интерфейсом.

Контрольные вопросы

1. Что такое структурированное программирование?
2. Что такое функция? Как описывается функция в *Python*?
3. Зачем нужны функции?
4. Для чего используется оператор *return* в функциях? Как вернуть из функции несколько значений?
5. Чем формальные параметры отличаются от фактических?
6. Что такое позиционные параметры?
7. Что такое параметры по умолчанию?
8. Чем отличается глобальная переменная от локальной?
9. Зачем применяются рекурсивные алгоритмы?
10. В чем заключается смысл принципа единственной ответственности (*Single Responsibility Principle*) при грамотной разработке единиц программного кода (в частности, функций)?

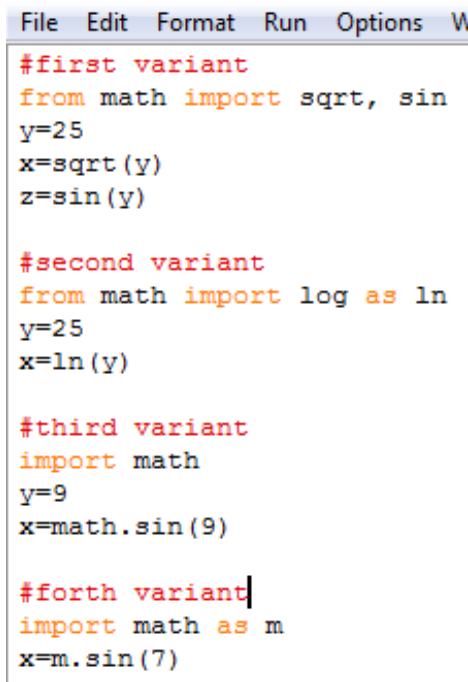
7. МОДУЛИ И ПАКЕТЫ

7.1. Модули и пакеты в *Python*

Функции, отвечающие за решение задач, принадлежащих к одной области, объединяют в модули, а модули – в пакеты.

Модули могут быть как стандартными (математический модуль `math`, модуль случайных чисел `random`) так созданными пользователем. В обоих случаях работа с функциями таких модулей организуется по одинаковым принципам: модуль (или функции модуля) необходимо импортировать.

Примеры импортирования, вызова функций модуля `math` с присвоением псевдонимов приведены на рис. 7.1.



```
File Edit Format Run Options W
#first variant
from math import sqrt, sin
y=25
x=sqrt(y)
z=sin(y)

#second variant
from math import log as ln
y=25
x=ln(y)

#third variant
import math
y=9
x=math.sin(9)

#forth variant
import math as m
x=m.sin(7)
```

Рис. 7.1. Модуль *math*

7.2. Лабораторная работа 7

«Модули и пакеты»

Цель работы: закрепить умения и навыки в создании структурированных программ, основанных на функциях, научиться группировать функции в модули и подключать их к основной программе.

Основное задание

Модернизировать программы и оптимизировать алгоритмы решения основных и дополнительных заданий лабораторной работы 6, группируя основные функции в отдельных модулях.

Требования к выполнению

1. Программа должна быть снабжена комментариями на английском языке, в которых необходимо указать краткое предназначение программы, номер лабораторной работы и ее название, версию программы, Ф.И.О. разработчика и дату разработки.

2. В программах, где это необходимо, предусмотреть возможность ее повторного выполнения, а также защиты от ввода некорректных данных (предусмотреть «защиту от дурака»). Для этих вещей рекомендуется разработать отдельные функции.

3. При разработке программ рекомендуется придерживаться *принципа единственной ответственности*.

4. Каждая программа должна быть снабжена дружелюбным и интуитивно понятным интерфейсом.

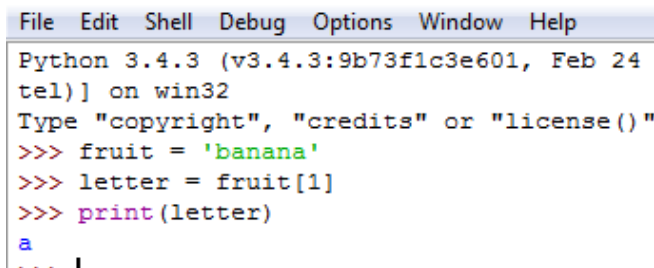
Контрольные вопросы

1. Базовые сведения о модулях.
2. Создание и использование модулей.
3. Использование оператора `import`.
4. Создание псевдонимов импортируемых функций.
5. Использование связки операторов `from/import`.
6. Поиск модулей. Переменная среды окружения `PYTHONPATH` и ее синтаксис.
7. «Компилирование» модулей.
8. Пространство имен и область видимости переменных.
9. Описать синтаксис встроенной функции `dir()`.
10. Создание и использование пакетов.

8. СТРОКИ И ОБРАБОТКА ТЕКСТОВОЙ ИНФОРМАЦИИ

8.1. Строки в *Python* и обработка текстовой информации

Строка – это неизменяемая (*immutable*) последовательность символов. Доступ к одному символу можно получить с помощью оператора квадратных скобок (рис. 8.1).

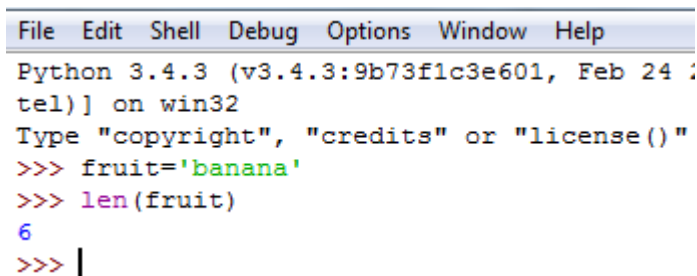


```
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24
tel)] on win32
Type "copyright", "credits" or "license()"
>>> fruit = 'banana'
>>> letter = fruit[1]
>>> print(letter)
a
... |
```

Рис. 8.1. Доступ к одному символу

Выражение в скобках называется *индексом* (*index*). Индекс – это смещение от начала строки; смещение для первого символа – нуль. Таким образом, нумерация в строках начинается с нуля. В качестве индекса можно использовать любое целочисленное выражение, включая переменные или операторы. Можно использовать отрицательные индексы, которые считаются с конца. `fruit[-1]` выдаст последний символ, `fruit[-2]` – второй с конца строки и так далее.

На рис. 8.2 приводится встроенная функция `len()`, которая возвращает число символов в строке.



```
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24
tel)] on win32
Type "copyright", "credits" or "license()"
>>> fruit='banana'
>>> len(fruit)
6
>>> |
```

Рис. 8.2. Встроенная функция `len()`

Обход строки с помощью цикла

Множество алгоритмов включают посимвольную обработку строк, которая, как правило, начинается с начала строки. Такая обработка называется *обходом* (*traversal*). Обход может осуществляться с помощью различных циклов.

На рис. 8.3 представлен обход с помощью цикла `while`.

```
>>> fruit='banana'
>>> index=0
>>> while index<len(fruit):
    letter=fruit[index]
    print(letter)
    index+=1

b
a
n
a
n
a
```

Рис. 8.3. Обход с помощью цикла `while`

Когда позиция символа не важна, а важно только его значение, то предпочтительнее использовать обход с помощью цикла `for` (рис. 8.4).

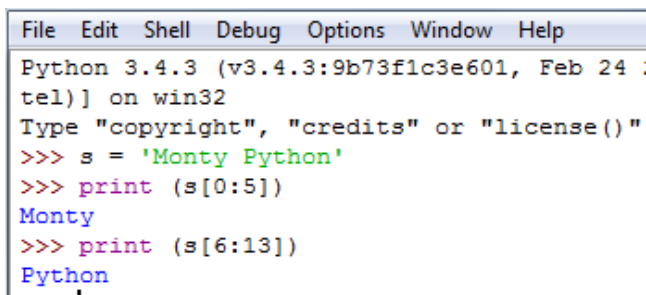
```
>>> fruit='banana'
>>> for char in fruit:
    print (char)

b
a
n
a
n
a
.
```

Рис. 8.4. Обход с помощью цикла `for`

Срез строки

Срезом (slice) называется часть строки. Оператор `[n:m]` возвращает часть строки от n -го символа до m -го, включая первый, но исключая последний. Если опустить первый индекс, то срез будет начинаться с начала строки. Если опустить второй – срез завершится в конце строки (рис. 8.5).

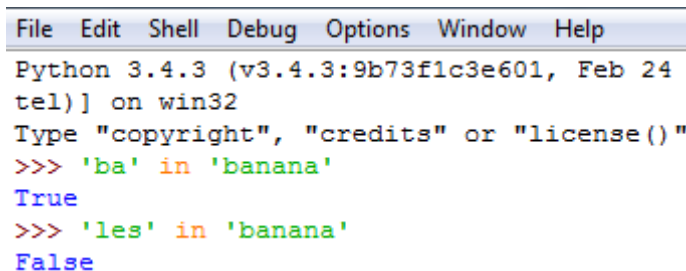


```
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 :
tel)] on win32
Type "copyright", "credits" or "license()"
>>> s = 'Monty Python'
>>> print (s[0:5])
Monty
>>> print (s[6:13])
Python
```

Рис. 8.5. Срез строки

Если первый индекс не меньше второго, то результатом будет *пустая строка*, если первый и второй индексы отсутствуют – выводится вся строка целиком.

Логический оператор `in` принимает две строки и возвращает `True`, если первая является подстрокой второй (рис. 8.6).



```
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24
tel)] on win32
Type "copyright", "credits" or "license()"
>>> 'ba' in 'banana'
True
>>> 'les' in 'banana'
False
```

Рис. 8.6. Логический оператор `in`

В *Python* можно сравнивать строки (рис. 8.7). Все буквы верхнего регистра имеют приоритет перед буквами нижнего, буквы – перед цифрами. При проверке равенства строк можно преобразовать строку в стандартный формат, например, нижний регистр.

```

File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24
tel)] on win32
Type "copyright", "credits" or "license()"
>>> 'apple' < 'banana'
True

```

Рис. 8.7. Сравнение строк

Строковые методы

Строка является объектом в *Python*. Объекты содержат данные (фактически саму строку) и методы, которые являются функциями, встроенными в объект и доступными для любого экземпляра (*instance*) объекта. Функция `dir()` выводит список доступных методов для объекта, вводимого в качестве параметра (рис. 8.8).

```

>>> dir('banana')
['_add_', '_class_', '_contains_', '_delattr_',
'_dir_', '_doc_', '_eq_', '_format_', '_ge_',
'_getattr_', '_getitem_', '_getnewargs_',
'_gt_', '_hash_', '_init_', '_iter_', '_le_',
'_len_', '_lt_', '_mod_', '_mul_', '_ne_',

```

Рис. 8.8. Функция `dir()`

Метод вызывается аналогично функции, принимает аргументы и возвращает значение. Однако метод и функция различаются синтаксисом: метод вызывается путем добавления имени метода к имени переменной с использованием точки в качестве разделителя. Синтаксис отдельных методов приводится в табл. 8.1.

Таблица 8.1

Строковые методы

| № | Метод | Описание |
|----|--------------------------|---|
| 1. | <code>str.upper()</code> | Преобразование строки <code>str</code> к верхнему регистру. |
| 2. | <code>str.lower()</code> | Преобразование строки <code>str</code> к нижнему регистру. |

| | | |
|-----|---|---|
| 3. | <code>str.find(str2,n)</code> | Возвращает индекс, с которого начинается вхождение строки <code>str2</code> в строку <code>str</code> . Необязательный параметр <code>n</code> – индекс, с которого начинается поиск вхождения. |
| 4. | <code>str.strip()</code> | Устранение лишних пробелов. |
| 5. | <code>str.capitalize()</code> | Преобразовывает из строчной в прописную первую букву строки <code>str</code> . |
| 7. | <code>str.count(str,beg=0, end=len(string))</code> | Подсчитывает количество вхождений подстроки <code>str</code> в оригинальной строке (<code>beg</code> – стартовая позиция в строке поиска, <code>end</code> – конечная позиция) |
| 9. | <code>str.endswith(suffix, beg=0, end=len(string))</code> | Возвращает булевское значение <code>True</code> , если оригинальная строка заканчивается последовательностью <code>suffix</code> , иначе возвращается <code>False</code> . |
| 10. | <code>str.startswith(suffix, beg=0, end=len(string))</code> | Возвращает булевское значение <code>True</code> , если оригинальная строка начинается с последовательности <code>suffix</code> , иначе возвращается <code>False</code> . |

8.2. Лабораторная работа 8

«Строки и обработка текстовой информации»

Цель работы: приобрести навыки работы с Python строками и закрепить их на примере разработки интерактивных приложений.

Основное задание

Написать программу «Анаграммы» («Anagrams»). Суть игры заключается в следующем: формируется группа слов в виде кортежа (*tuple*), компьютер случайным образом выбирает одно из слов, и случайным образом переставляет в нем буквы, а затем представляет пользователю (игроку). Цель игрока – угадать выбранное и «перемешанное» компьютером слово.

Индивидуальное задание

В соответствии с заданием своего варианта выполнить задания из прил. С.

Дополнительное задание

Модифицировать программу «Анаграммы» («Anagrams») таким образом, чтобы в ней были уровни прохождения игры. К примеру, в игре «Анаграммы» можно реализовать 10 уровней, где на первых уровнях предлагаются простые слова. При этом от уровня к уровню слова становятся длиннее, и растет их уровень сложности. Дополнительно можно добавиться к игре ограниченное количество подсказок и попыток угадывания.

Требования к выполнению

1. Программа должна обязательно быть снабжена комментариями на английском языке, в которых необходимо указать краткое предназначение программы, номер лабораторной работы и название, версию программы, Ф.И.О. разработчика и дату разработки.
2. В программах, предусмотреть возможность ее повторного выполнения, а также защиты от ввода некорректных пользовательских данных. Для этого рекомендуется разработать отдельные функции.
3. Разработанные основные функции бизнес-логики необходимо разместить в отдельном модуле, а затем подключить в другом модуле, где будет происходить тестирование данных функций.
4. При разработке программ рекомендуется придерживаться принципа единственной ответственности.
5. Каждая программа должна быть снабжена дружелюбным и интуитивно понятным интерфейсом.

Контрольные вопросы

1. Чем характеризуется строковый тип данных в *Python*?
2. Какие есть способы объявления строк в *Python*?
3. Что такое неизменяемость строк?
4. Зачем нужна индексация строк, и как ее использовать?
5. Зачем и как используются срезы строк?
6. Какие основные операторы используются для работы со строками?
7. Какие основные встроенные функции класса `str` используются для работы со строками?
8. Как объявить шаблонную строку, и какие спецификаторы типов (операторы форматирования) в этой строке можно использовать?

9. Что необходимо прописать в исходном коде программы, если нужно использовать различные кодировки для отображения текстовой информации?

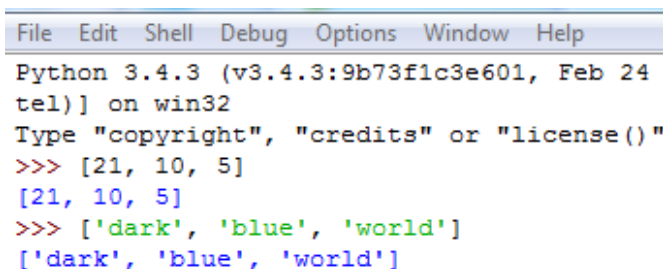
10. Как создать всплывающую подсказку с описанием вызываемой функции?

9. КОРТЕЖИ, СПИСКИ, СЛОВАРИ И МНОЖЕСТВА

9.1. Кортежи, списки, словари и множества в *Python*

Списки

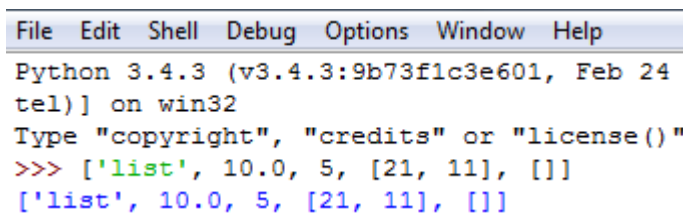
Список (list) – изменяемая последовательность значений. Элементы списка заключаются в квадратные скобки. Элементы списка могут быть разного типа (рис. 9.1).



```
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24
tel)] on win32
Type "copyright", "credits" or "license()"
>>> [21, 10, 5]
[21, 10, 5]
>>> ['dark', 'blue', 'world']
['dark', 'blue', 'world']
```

Рис. 9.1. Списки целых чисел, строк

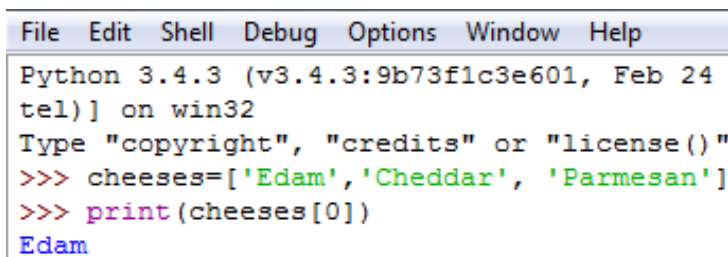
Список внутри другого списка является *вложенным (nested)*. Список без элементов, называется *пустым*. Пустой список создается с помощью пустых квадратных скобок [] (рис. 9.2).



```
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24
tel)] on win32
Type "copyright", "credits" or "license()"
>>> ['list', 10.0, 5, [21, 11], []]
['list', 10.0, 5, [21, 11], []]
```

Рис. 9.2. Вложенные и пустые списки

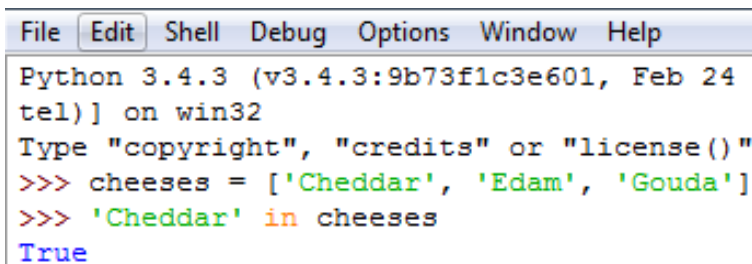
Синтаксис для доступа к элементам списка – операторы квадратных скобок. Выражение внутри скобок определяют индекс. Индексация начинается с нуля. Любое целочисленное выражение можно использовать в качестве индекса. Если индекс имеет отрицательное значение, то отсчет элементов ведется в обратном направлении от конца списка (рис. 9.3).

A screenshot of a Python 3.4.3 IDLE window. The menu bar at the top includes File, Edit, Shell, Debug, Options, Window, and Help. The main text area shows the following code: Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 tel)] on win32, Type "copyright", "credits" or "license()", >>> cheeses=['Edam', 'Cheddar', 'Parmesan'], >>> print(cheeses[0]), and the output Edam.

```
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24
tel)] on win32
Type "copyright", "credits" or "license()"
>>> cheeses=['Edam', 'Cheddar', 'Parmesan']
>>> print(cheeses[0])
Edam
```

Рис. 9.3. Операторы квадратных скобок

Оператор «+» объединяет списки. Оператор «*» повторяет список несколько раз. Оператор среза для списков работает похожим образом, как для строк. Оператор in возвращает результат вхождения одного списка в другой список (рис. 9.4).

A screenshot of a Python 3.4.3 IDLE window. The menu bar at the top includes File, Edit, Shell, Debug, Options, Window, and Help. The main text area shows the following code: Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 tel)] on win32, Type "copyright", "credits" or "license()", >>> cheeses = ['Cheddar', 'Edam', 'Gouda'], >>> 'Cheddar' in cheeses, and the output True.

```
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24
tel)] on win32
Type "copyright", "credits" or "license()"
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Cheddar' in cheeses
True
```

Рис. 9.4. Оператор in

Обход списка

Для обхода списка удобно использовать цикл for (рис. 9.5).

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> for cheese in cheeses:
    print (cheese)

Cheddar
Edam
Gouda
```

Рис. 9.5. Обход списка с помощью цикла for

Можно использовать конструкцию с индексами (рис. 9.6).

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> for i in range(len(cheeses)):
    print(i, '-', cheeses[i])

0 - Cheddar
1 - Edam
2 - Gouda
```

Рис. 9.6. Обход списка

Методы списков

В табл. 9.1 приводятся некоторые методы списков.

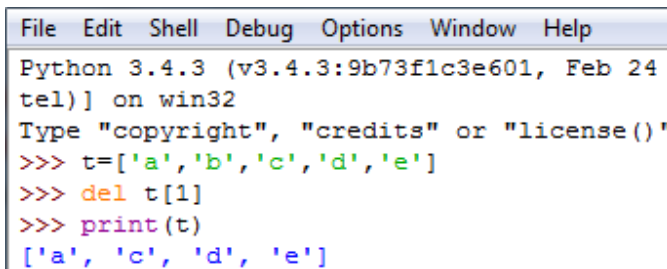
Таблица 9.1

Методы списков

| № | Метод | Описание |
|----|------------------------------------|--|
| 1. | <code>list1.append(element)</code> | Добавляет элемент <code>element</code> в конец списка <code>list1</code> . |
| 2. | <code>list1.append(list2)</code> | Добавляет элементы списка <code>list2</code> в список <code>list1</code> . |
| 3. | <code>list1.sort()</code> | Сортировка по возрастанию. |

| | | |
|----|------------------------------------|---|
| 4. | <code>list1.pop(index)</code> | Удаление элемента с индексом <code>index</code> . Возвращается удаленный элемент. |
| 5. | <code>list1.remove(element)</code> | Удаляет элемент <code>element</code> из списка <code>list1</code> . |

Для удаления значений списков можно воспользоваться оператором `del` (рис. 9.7).



```

File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24
tel) on win32
Type "copyright", "credits" or "license()"
>>> t=['a','b','c','d','e']
>>> del t[1]
>>> print(t)
['a', 'c', 'd', 'e']

```

Рис. 9.7. Оператор `del`

Списки и функции

Существуют встроенные функции для работы со списками: `len()` – возвращает длину, `max()` (`min()`) – поиск экстремальных элементов, `sum()` – суммирование элементов.

Списки и строки

Функция `list()` преобразует строку в список символов (рис. 9.8). Поскольку `list()` – это имя встроенной функции, следует избегать его использование в качестве имени переменной. Функция `list()` разбивает строку на отдельные буквы.

```

>>> s = 'spam'
>>> list(s)
['s', 'p', 'a', 'm']

```

Рис. 9.8. Функция `list()`

Функция `split()` разбивает строку на слова (рис. 9.9).

```
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24
tel)] on win32
Type "copyright", "credits" or "license()"
>>> s = 'pining for the fjords'
>>> t=s.split()
>>> print(t)
['pining', 'for', 'the', 'fjords']
```

Рис. 9.9. Функция `split()`

При вызове `split()` можно передать аргумент, который задает разделитель, вместо пробела по умолчанию (рис. 9.10).

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> s.split(delimiter)
['spam', 'spam', 'spam']
```

Рис. 9.10. Функция `split()` с разделителем `delimiter`

Функция `join()` объединяет элементы списков (рис. 9.11).

```
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24
tel)] on win32
Type "copyright", "credits" or "license()"
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> delimiter.join(t)
'pining for the fjords'
```

Рис. 9.11. Функция `join()`

Словари

Словарь (dictionary) похож на список, но имеет более широкие возможности. В списке индекс имеет целочисленное значение, в словаре может быть любого типа. `dict()` создает словарь без записей (рис. 9.12). Ключ 'one' отображается в значение 'uno'.

```

>>> eng2sp = dict()
>>> print(eng2sp)
{}
>>> eng2sp['one'] = 'uno'
>>> print(eng2sp)
{'one': 'uno'}

```

Рис. 9.12. Создание словаря

Для добавления записей в словарь, можно использовать квадратные скобки. Словарь с тремя записями представлен на рис. 9.13.

```

File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) |
tel)] on win32
Type "copyright", "credits" or "license()" for more informa
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}

```

Рис. 9.13. Словарь с тремя записями

Вывод словаря на разных компьютерах может дать разный результат, так как порядок пар ключ-значение не всегда совпадает. Функция `len()` возвращает число пар ключ-значение. Оператор `in` сообщает о похожих ключах. Метод `keys()` возвращает множество ключей словаря в виде списка, а метод `values()` возвращает в виде списка множество значений словаря (рис. 9.14).

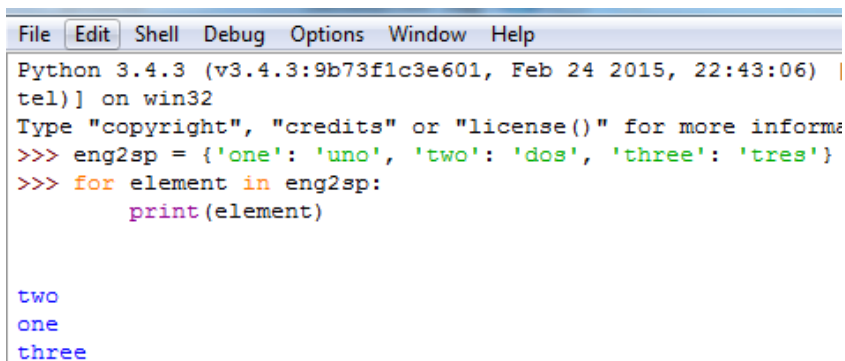
```

File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06)
tel)] on win32
Type "copyright", "credits" or "license()" for more inform
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
>>> eng2sp.keys()
dict_keys(['two', 'three', 'one'])
>>> eng2sp.values()
dict_values(['dos', 'tres', 'uno'])

```

Рис. 9.14. Метод `keys()`

Для обхода словаря можно использовать `for` (рис. 9.15).

A screenshot of a Python 3.4.3 Shell window. The window has a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The shell prompt is 'Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [tel] on win32'. The user enters the command 'Type "copyright", "credits" or "license()" for more inform:' followed by '>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}'. Then the user enters a for loop: '>>> for element in eng2sp: print(element)'. The output shows the keys of the dictionary: 'two', 'one', and 'three' on separate lines.

```
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [tel] on win32
Type "copyright", "credits" or "license()" for more inform:
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
>>> for element in eng2sp:
    print(element)

two
one
three
```

Рис. 9.15. Обход словаря с помощью цикла *for*

9.2. Пример выполнения индивидуального задания

Пусть дана последовательность действительных чисел объема n . Необходимо:

- поменять местами наибольший и наименьший (экстремальные) элементы (если их несколько, то поменять первые из найденных);
- найти сумму и произведение всех положительных элементов.

Код итоговой программы представлен на рис. 9.16.

Замечание. Коды функций для обмена экстремальных элементов и поиска суммы и произведения положительных элементов не приводятся, их необходимо уметь разрабатывать самостоятельно!

```

# Example of program working:
# A sequence of real numbers (n = 5): 1.2, -2.5, 0.5, 3.0, -0.5
# Result:
#   original sequence: [1.2, -2.5, 0.5, 3.0, -0.5]
#   final sequence:   [1.2, 3.0, 0.5, -2.5, -0.5]
#   max = 3.0
#   min = -2.5
#   sum = 4.7
#   multiplication = 1.8

def main():
    import os
    import sys
    import module_for_task_9 as mdl

    lst = []
    n = int(input("Input the size of the sequence: "))

    print("Input real elements of the sequence:")

    for _ in range(n):
        lst.append(float(input()))

    if sys.platform == 'win32':
        os.system('cls')
    else:
        os.system('clear')

    print("Result of task 9:")

    print("\toriginal sequence: ", end="\t")
    print(lst)

    mdl.change_extreme_elements(lst)

    print("\tfinal sequence: ", end="\t")
    print(lst)

    print("\tmax element: " + str(mdl.get_max_element_of_list(lst)))
    print("\tmin element: " + str(mdl.get_min_element_of_list(lst)))
    print("\tsum = " + str(mdl.get_sum_of_positive_elements(lst)))
    print("\tmultiplication = " +
          str(mdl.get_multiplication_of_positive_elements(lst)))

main()
input("\n\nPlease, press Enter for exit...")

```

Подключаем модуль с бизнес-функциями и создаём псевдоним для сокращённого использования имени модуля

Объявляем пустой список и по очереди добавляем с использованием метода *append()* элементы в список

Производим очистку консоли в зависимости от операционной системы

Вывод элементов списка

Вызов соответствующих функций из подключённого модуля

Не забываем про эту хитрость ☺

Рис. 9.16. Код тестового модуля

9.3. Лабораторная работа 9

«Кортежи, списки, словари и множества»

Цель работы: приобрести навыки работы со стандартными встроенными высокоуровневыми типами данных в Python и закрепить их на примере разработки интерактивных приложений.

Основное задание

Разработать интерактивную программу, которая будет моделировать игру «Виселица» («Hangman»). Компьютер загадывает слово и выводит на консоль количество подчеркиваний, равное числу букв в загаданном слове. Пользователь начинает вводить буквы, чтобы отгадать слово. Если буква есть в слове, компьютер вписывает ее на свое место в слово, а если нет, компьютер рисует в консоли один элемент импровизированной виселицы (к примеру: стойка, перекладина, веревка, голова, туловище, две руки, две ноги). Дополнительно выводятся буквы, которые уже вводились.

Если игрок не успел угадать слово раньше, чем компьютер нарисовал полностью виселицу, то игрок считается проигравшим. Если игрок успевает угадать слово, то он выигрывает.

Индивидуальное задание

В соответствии с заданием своего варианта выполнить задания из прил. D и E.

Дополнительное задание

Модифицировать программу «Виселица» таким образом, чтобы в ней были уровни сложности и (или) подсказки.

Требования к выполнению

1. Каждое задание оформить в виде отдельной бизнес-функции. Все функции необходимо сгруппировать в одном модуле или разнести по отдельным модулям, согласно их логике их работы.
2. Размерность списка и матрицы задается пользователем.
3. Предусмотреть два способа инициализации последовательности и матрицы: с помощью генератора случайных чисел и пользовательского ввода. Оформить способы инициализации в виде отдельных функций, которые на вход принимают последовательность или

матрицу для инициализации, и сгруппировать эти функции в отдельный модуль от основной функции программы.

4. В программах предусмотреть возможность повторного выполнения и защиту от ввода некорректных пользовательских данных. Для этих целей рекомендуется разработать отдельные функции.

5. При разработке программ рекомендуется придерживаться принципа единственной ответственности.

6. Программа должна обязательно быть снабжена комментарием на английском языке, в которых необходимо указать краткое предназначение программы, номер лабораторной работы и название, версию программы, Ф.И.О. разработчика и дату разработки.

7. Программа должна быть снабжена дружелюбным и интуитивно понятным интерфейсом.

Контрольные вопросы

1. Приведите примеры объявления каждого из высокоуровневых типов данных.

2. Объясните понятие «распаковка последовательности».

3. Какое главное отличие списков от кортежей? Когда лучше использовать кортежи, а когда – списки?

4. Что такое распределенные ссылки?

5. Как получить несколько ссылочных переменных на один и тот же список, а как получить полную копию списка?

6. Зачем нужна индексация элементов высокоуровневых типов? Как ее использовать?

7. Как получить доступ к элементам словаря? Можно ли использовать индексацию для словарей?

8. Зачем и как используются срезы?

9. Какие операторы и встроенные функции используются для работы с кортежами, списками, словарями и множествами? Приведите сравнительную таблицу.

10. Какие методы есть у каждого из классов, отвечающих за каждый высокоуровневый тип данных (*tuple*, *list*, *dict* и *set*)? Опишите наиболее востребованные.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Доусон, М. Программируем на Python / М. Доусон – СПб.: Питер, 2014. – 416 с.
2. Лутц, М. Изучаем Python / М. Лутц. – 4-е издание.; пер. с англ. – СПб.: Символ-Плюс, 2011. – 1280 с.
3. Лутц, М. Программирование на Python / М. Лутц. – 4-е издание.; пер. с англ. – СПб.: Символ-Плюс, 2011. – Т. 1. – 992 с.
4. Лутц М. Программирование на Python / М. Лутц. – 4-е издание.; пер. с англ. – Пб.: Символ-Плюс, 2011. – Т. 2. – 992 с.
5. Прохоренок, Н. А. Python 3 и PyQt. Разработка приложений / Н. А. Прохоренок. – СПб.: БХВ-Петербург, 2012. – 704 с.
6. Хахаев, И. А. Практикум по алгоритмизации и программированию на Python / И. А. Хахаев – М.: Альт Линукс, 2010. – 126 с.
7. Шапошникова, С. Основы программирования на Python [Электронный ресурс] / С. Шапошникова. – Режим доступа: <http://youglinux.info>. – Дата доступа: 20.12.2016.
8. Северанс, Ч. Р. Введение в язык программирования Python [Электронный ресурс] / Ч. Р. Северанс. – Режим доступа: <http://pycode.ru/edu/why-python>. – Дата доступа: 20.12.2016.
9. Pythonic way [Электронный ресурс]. – Режим доступа: <http://pythonicway.com>. – Дата доступа: 20.12.2016.