

Министерство науки и высшего образования Российской Федерации

Южно-Российский государственный
политехнический университет (НПИ) имени М.И. Платова

С.Н. Широбокова, А.А. Кацупеев, А.В. Сулыз

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ *PYTHON*

**Учебное пособие
для лабораторных занятий**

**Новочеркасск
ЮРГПУ(НПИ)
2020**

УДК 004.43 (075.8)
ББК 32.973-018.1я73
Ш64

Рецензенты: **А.Н. Ткачев**, заведующий кафедрой «Прикладная математика» ЮРГПУ(НПИ), доктор технических наук, профессор
Д.В. Гринченков, декан факультета информационных технологий и управления, заведующий кафедрой «Программное обеспечение вычислительной техники» ЮРГПУ(НПИ), кандидат технических наук, доцент.

Широбокова С.Н., Кацупеев А.А., Сулыз А.В.

Ш64 Программирование на языке *Python* : учебное пособие для лабораторных занятий / С.Н. Широбокова, А.А. Кацупеев, А.В. Сулыз; Южно-Российский государственный политехнический университет (НПИ) имени М.И. Платова. – Новочеркасск: ЮРГПУ(НПИ), 2020.– 104с.

ISBN 978-5-9997-0725-3

Учебное пособие содержит краткую теоретическую часть об истории развития языка программирования *Python*, его наиболее востребованных расширениях и функциональных возможностях. Рассмотрены причины популярности языка, сферы его основной деятельности в настоящее время, а также принципы его работы и технологий, на которых он основан. Пособие включает материал для лабораторных работ по дисциплине «Программирование на языке *Python*», в рамках которых изучается синтаксис языка программирования *Python*, его использование для решения задач анализа, обработки данных и их визуализации, а также создания десктопных и веб-приложений.

Предназначено для студентов вузов, обучающихся по направлениям подготовки «Прикладная информатика» (бакалавриат) и «Информационные системы и технологии» (бакалавриат). Пособие может быть полезно студентам других компьютерных направлений подготовки, изучающим основы программирования на языке *Python*.

УДК 004.43 (075.8)
ББК 32.973-018.1я73

ISBN 978-5-9997-0725-3

© Южно-Российский государственный
политехнический университет
(НПИ) имени М.И. Платова, 2020

Содержание

Введение	4
История создания	6
Дзен <i>Python</i>	8
Концепция работы и внутренняя структура языка <i>Python</i>	9
Популярность <i>Python</i>	11
Популярные инструменты разработки на <i>Python</i>	13
Сферы применения <i>Python</i>	14
Функциональное программирование в <i>Python</i>	15
<i>PEP 8</i>	19
Модули и пакеты <i>Python</i>	23
<i>Python Virtual Environments</i>	24
<i>Лабораторная работа №1.</i> Работа в <i>IDLE</i> . Использование арифметических операций. Создание списков и словарей. Работа с циклами.	26
<i>Лабораторная работа №2.</i> Работа с основными встроенными функциями	36
<i>Лабораторная работа №3.</i> Работа с итераторами, генераторами. Работа с генераторными выражениями.	40
<i>Лабораторная работа №4.</i> Работа с основными модулями.	46
<i>Лабораторная работа №5.</i> Работа с файлами. Разработка синтаксического анализатора. Вывод форматированных данных в формате <i>JSON</i> .	52
<i>Лабораторная работа №6.</i> Разработка приложения работы с базой данных.	59
<i>Лабораторная работа №7.</i> Разработка приложения с использованием ООП.	64
<i>Лабораторная работа №8.</i> Обработка изображений с применением библиотеки <i>PIL</i>	71
<i>Лабораторная работа №9.</i> Разработка <i>GUI</i> приложения с помощью графических библиотек.	76
<i>Лабораторная работа №10.</i> Визуализация результатов работы математических алгоритмов с использованием <i>numpy</i> и <i>matplotlib</i> .	82
<i>Лабораторная работа №11.</i> Основы работы с веб-фреймворком <i>Django</i> .	90
Заключение	102
Библиографический список	103

ВВЕДЕНИЕ

Python – это интерпретируемый язык программирования общего назначения. Основные преимущества данного языка программирования заключаются в его простоте освоения и универсальности, структурированности и читабельности кода. Семантическое ядро имеет очень удобную структуру, а широкий перечень встроенных библиотек позволяет применять внушительный набор полезных функций и возможностей. При создании этого языка программирования авторы старались взять лучшее из различных платформ для разработчиков. Фактически *Python* представляет собой своеобразную «смесь» удачных решений более чем из 8 различных языков. *Python* является самым быстрорастущим языком программирования за последние несколько лет, согласно исследованию *StackOverflow* в 2019 г.

Python используется почти в каждом среднем или крупном проекте, если не как основной инструмент разработки, то как инструмент для создания прототипа или написания какой-то его части. *Python* может применяться почти в любой задаче, однако основными сферами его применения являются:

- веб-разработка;
- машинное обучение, анализ данных и визуализация;
- автоматизация процессов или скриптинг.

Его можно также применять для написания игр, используя библиотеку *PyGame*, однако её популярность не слишком велика. Существует немало десктопных приложений, разработанных на *Python*. Популярным решением для разработки десктопных приложений является использование языка с графическим фреймворком *Qt* (*PyQt*).

Python используют множество компаний и разработчиков по всему миру:

- *Spotify* и *Amazon* используют его в задачах анализа данных;
- *Walt Disney* использует его как скриптовый язык для создания анимации;
- *Youtube* и *Instagram* написаны на *Python*;

– рекомендательный сервис *Netflix* также написан на *Python*;

– *NASA*, *Los Alamos*, *Fermilab*, *JPL* используют *Python* для научных вычислений;

– Агентство национальной безопасности США – для шифрования и анализа разведданных;

– *ESRI* – как инструмент настройки геоинформационных программ;

– *JPMorgan Chase*, *UBS*, *Getco* и *Citadel* – для прогнозирования финансового рынка;

– *iRobot* – для разработки коммерческих роботизированных устройств.

Кроме того, его используют в *Positive Technologies*, *Houdini*, *Facebook*, *Yahoo*, *Red Hat*, *Dropbox*, *Pinterest*, *Quora*, *Mail.ru* и «Яндексе».

Python – это язык программирования, востребованный сегодня и с большим потенциалом в будущем. Рынок труда нуждается в квалифицированных специалистах со знаниями *Python*.

В настоящее время *Python* является одним из самых популярных и простых в освоении языков программирования, получивших широкое распространение. Код, написанный на *Python*, легко читать и писать. Язык включает огромное количество встроенных в него функциональных возможностей и библиотек, которые позволяют, не прибегая к сторонним инструментам, разрабатывать полноценные десктопные приложения, приложения для научных исследований и прочее. При этом широкая популярность языка и любовь среди разработчиков позволили ему обрести огромное количество внешних библиотек и фреймворков, которые, расширяя возможности языка, позволяют создавать игры, веб-приложения, мобильные приложения, применять технологии машинного обучения и анализа данных и многое другое. Также *Python* за счет его простоты и удобства проникает в сферу программирования микроконтроллеров. Уже сейчас существует стабильная реализация *Python*, называемая *Micro Python*, которая оптимизирована для работы на 32-битных *ARM* микроконтроллерах. Негативной же стороной языка является скорость его работы, что не позволяет ему полноценно использоваться в некоторых сферах. Однако,

хоть *Python* и не может применяться в некоторых сферах, его часто используют в качестве написания прототипов программ, которые впоследствии переносят на более быстрые языки, такие как *C/C++* или *Java*.

В пособие включена краткая информация об истории развития языка программирования *Python*, его наиболее востребованных расширениях и функциональных возможностях. Рассмотрены причины популярности языка, сферы его основной деятельности в настоящее время, а также принципы его работы и технологий, на которых он основан. Материал для лабораторных работ дает представление об основном синтаксисе языка программирования *Python*, его использовании для решения задач анализа, обработки данных и их визуализации, создания десктопных и веб-приложений.

Учебное пособие включает весь необходимый теоретический и практический материал, а также варианты заданий для организации проведения лабораторных занятий по дисциплине «Программирование на языке *Python*».

ИСТОРИЯ СОЗДАНИЯ

Первый релиз *Python* был опубликован в феврале 1991 года, автором которого стал сотрудник голландского института *CWI* Гвидо Ван Россум, который взял за основу язык *ABC*.

Стоит отметить тот факт, что название языка не относится к виду змей. Гвидо Ван Россум на момент создания языка был поклонником сериала «Воздушный цирк Монти Пайтона». Имя главного героя сериала Монти Пайтона и послужило названием языка.

Python появился сравнительно позднее других языков и создавался под влиянием множества других языков программирования. Из каждого языка *Python* позаимствовал лучшее и полезное, что они предлагали. Например, влияние на создание оказали следующие языки программирования [1]:

- *ABC*: отступы для группировки операторов и высокоуровневые структуры данных;
- *Smalltalk*: объектно-ориентированное программирование;

- C, C++: некоторые базовые синтаксические конструкции;
- *Modula-3*: модули и пакеты;
- *SETL* и *Haskell*: генераторные выражения (*list comprehensions*);
- *Java*: обработка исключений и другое.

Так же, как и приведенные выше, большая часть других особенностей *Python* не приносила что-то новое, так как была реализована уже в других языках.

В январе 1994 года увидел свет релиз первой версии языка *Python* 1.0, в котором были добавлены средства функционального программирования: *map*, *filter*, *reduce* и лямбда-исчисление.

В 2000 году был выпущен *Python* 2.0, поддержка которого прекратилась 1 января 2020 года. Начиная с версии 2.1 весь код, техническая документация и спецификации принадлежат некоммерческой организации *Python Software Foundation*, созданной в 2001 году по образцу *Apache Software Foundation*. В версии 2.2 в языке были объединены базовые типы и классы, которые создавали программисты-пользователи в одну иерархию, что сделало *Python* полностью объектно-ориентированным языком.

В 2008 году после долгого тестирования выпущена версия языка *Python* 3.0, в которой устранили многие недостатки архитектуры, при этом была реализована максимально возможная совместимость с версией 2.x. Основной целью разработки версии 3.0 было устранение фундаментальных изъянов в языке.

Несмотря на такие большие разрывы между версиями языка постоянно выходят подверсии. На момент написания данного материала актуальной версией является *Python* 3.8.

В настоящее время разработка языка ведется строго по процессу, регламентированному в *PEP* – предложений по развитию *Python*.

Python распространяется под лицензией открытого программного обеспечения, называемой *Python Software Foundation License (PSFL)*. Это *BSD*-подобная лицензия совме-

стимая с *GPL*. Однако в отличие от *GPL* лицензия *Python* позволяет вносить изменения в исходный код, а также создавать производственные работы, не открывая код. Стоит отметить, что лицензия *GPL* предоставляет пользователю право на копирование, модификацию и распространение программы с условием того, что пользователи всех производных программ получат эти права. Принцип «наследования» прав называется «копилефт», т.е. можно отметить, тот факт, что программы, реализованные под данной лицензией нельзя включать в проприетарное ПО. Лицензия *PSFL* в отличие от *GPL* не является копилефтной.

ДЗЕН PYTHON

Разработчики языка *Python* придерживаются определенной философии программирования, называемой «*The Zen of Python*» («Дзен Питона»). Его автором считается один из создателей языка Тим Петерс. Текст дзена выдается интерпретатором *Python* по команде *import this*. В целом он применим к программированию на всех языках.

Текст дзена:

- Красивое лучше, чем уродливое.
- Явное лучше, чем неявное.
- Простое лучше, чем сложное.
- Сложное лучше, чем запутанное.
- Плоское лучше, чем вложенное.
- Разреженное лучше, чем плотное.
- Читаемость имеет значение.
- Особые случаи не настолько особые, чтобы нарушать правила.
- При этом практичность важнее безупречности.
- Ошибки никогда не должны замалчиваться.
- Если они не замалчиваются явно.
- Встретив двусмысленность, отбрось искушение угадать.
- Должен существовать один и, желательно, только один очевидный способ сделать это.

- Хотя он поначалу может быть и не очевиден, если вы не голландец.
- Сейчас лучше, чем никогда.
- Хотя никогда зачастую лучше, чем прямо сейчас.
- Если реализацию сложно объяснить — идея плоха.
- Если реализацию легко объяснить — идея, возможно, хороша.
- Пространства имён — отличная штука! Будем делать их больше!

КОНЦЕПЦИЯ РАБОТЫ И ВНУТРЕННЯЯ СТРУКТУРА ЯЗЫКА *PYTHON*

Python является интерпретируемым языком программирования, который компилируется в байт-код и выполняется в собственной виртуальной машине *PVM*. Это позволяет языку быть кроссплатформенным, т.е. работать на любых операционных системах, для которых существует реализация интерпретатора языка.

Интерпретатор — это программа выполняющая процесс интерпретации.

Интерпретация — это процесс анализа, обработки и выполнения исходного кода программы. Интерпретаторы делятся на простой интерпретатор и интерпретатор компилирующего типа. В случае с первым интерпретация выполняется построчно при считывании исходного кода и при таком процессе обнаружить ошибки в программе можно только в момент обработки команды с ошибкой. Интерпретатор компилирующего типа представляет собой связку из компилятора, который переводит исходный текст программы в промежуточное представление, например, в байт-код и интерпретатор, который его исполняет (виртуальная машина).

Виртуальная машина (VM) — это абстрактная вычислительная машина, которая не зависит от аппаратного обеспечения и операционной системы. Реализацией *VM* является программный или программно-аппаратный симулятор *VM*, работающий на реальной аппаратуре, обычно поверх существующей опера-

ционной системы. Основная цель является в том, что программа, скомпилированная в код виртуальной машины, выполняется везде, где имеется реализация *VM*. Виртуальные машины определяют свой набор инструкций, абстрактный процессор для их исполнения, собственную систему памяти, интерфейс взаимодействия с прикладными библиотеками и операционной системой и др. Операционные системы и виртуальные машины схожи. Их задачей является абстрагирование приложения от подробностей аппаратного обеспечения. Они содержат схожие компоненты такие как управление памятью, потоки и процессы, ввод-вывод. Однако виртуальные машины ориентированы на язык или класс языков, которые стремятся не зависеть от аппаратуры, а операционные системы ориентированы на класс аппаратного обеспечения и стремятся не зависеть от языка.

Сам *Python* имеет несколько реализаций интерпретаторов. Эталонной реализацией является интерпретатор *CPython*, которая написана на языке программирования *C*. Данная реализация ведется разработчиками под руководством создателя языка Гвидо Ван Россума и, как правило, работает быстрее, устойчивее и лучше, чем альтернативные реализации. Альтернативных реализаций существует несколько:

- *Jython* – это реализация *Python* на языке *Java*. Программы, выполняющиеся в данной среде, могут одновременно использовать классы языков *Python* и *Java*;

- *IronPython* – это реализация *Python* для платформы *.NET* написанная на языке *C#*. Так же, как и с предыдущей реализацией среда имеет доступ к классам *C#* и *Python*;

- *PyPy* – это реализация *Python* написанная на *Python*. В него встроен *JIT*-компилятор, который позволяет приводить код на *Python* в машинный код вовремя выполнение программы.

- *StacklessPython* – это реализация *Python* написанная на языке *C*, целью которой является отказ от использования стандартного стека вызовов языка *C* в пользу собственного стека. Особенностью данной среды является использование микропотоков, которые позволяют избежать чрезмерного расхода системных ресурсов, которые присущи операционным системам.

При запуске любого скрипта, написанного на *Python*, происходит перевод исходного текста в байт-код для виртуальной машины осуществляемый компилятором. *Python* транслирует каждую инструкцию в исходном коде программы в группы инструкций байт-кода для повышения скорости выполнения программы, поскольку выполнение байт-кода происходит гораздо быстрее. После компиляции в байт-код рядом с исходным файлом скрипта создается файл с расширением «.рус».

При последующем запуске программы, если исходный код не был изменен, интерпретатор минует процесс компиляции и сразу запустит выполнение скомпилированного файла. Если же исходный текст скрипта был изменен, то процесс компиляции произойдет снова.

Однако в некоторых случаях, *Python* не сможет записать скомпилированный файл, это может возникнуть, например, при отсутствии чтения прав на запись. В этом случае скомпилированный байт-код просто загрузится в оперативную память и после завершения исполнения программы будет из неё удален.

ПОПУЛЯРНОСТЬ *PYTHON*

Исследования *StackOverflow* показали, что *Python* претендует на роль самого быстрорастущего основного языка программирования. Причиной такого бурного развития стало его применение в самых различных задачах, начиная от веб-разработки и заканчивая работой с данными и *DevOps*. Стоит отметить, что в последнее время приложения, разработанные на *Python*, становятся всё более распространёнными.

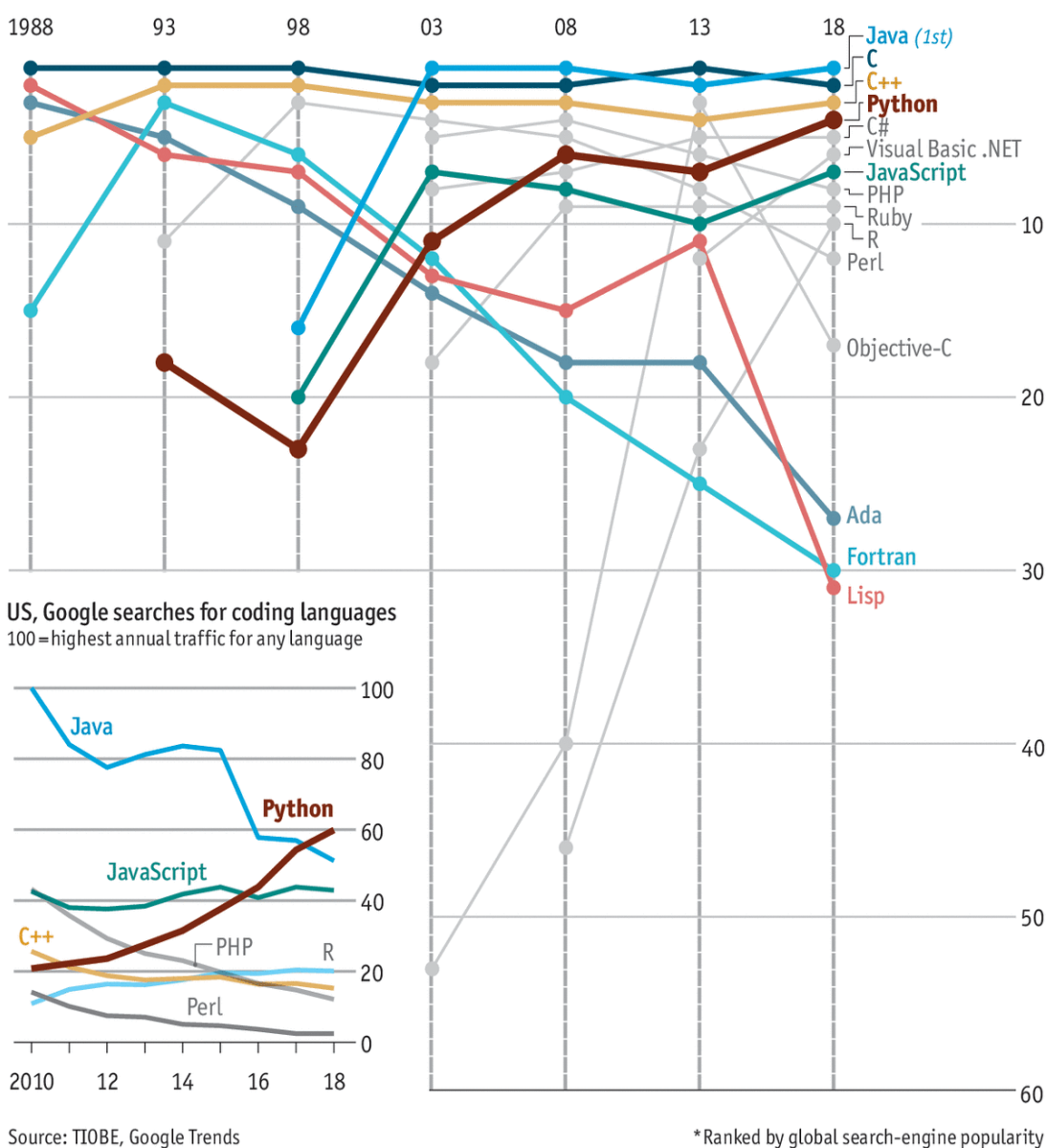
На текущий момент наиболее популярной сферой для использования *Python* является сфера анализа данных, машинного обучения и научных исследований. Статистика *StackOverflow* показывает, что областями использования *Python* являются электроника, производство, разработка программ, правительство и особенно язык популярен в учебных заведениях. Однако рост *Python* по отраслям происходит довольно равномерно. В совокупности это говорит о том, что анализ данных и машинное обучение получают распространение в различных типах

компаний, и *Python* становится общим и популярным решением для подобных задач [2].

Основными причинами популярности языка является его простота и универсальность. Синтаксис языка достаточно прост, что помогает легко учиться и читать программы, написанные на нем. Универсальность, демонстрируется на примерах его использования, например, ЦРУ использовало его для взломов, *Google* – для веб-сканирования, *Pixar* – для производства фильмов и *Spotify* – для рекомендаций песен.

Code of conduct

Ranking of programming languages*



The Economist

Рис. 1. График роста популярности *Python* [3]

Сам создатель языка говорил, что не собирался создавать языка для массового использования. Но почти три десятилетия спустя его изобретение обогнало почти всех своих конкурентов. Количество запросов по поиску «*Python*» утроилось с 2010 года, в то время как количество запросов для других основных языков программирования не изменилось [3]. На рис. 1 приведен график роста популярности языка *Python* с 1993 года.

ПОПУЛЯРНЫЕ ИНСТРУМЕНТЫ РАЗРАБОТКИ НА *PYTHON*

За все время своего существования для *Python* появилось огромное количество инструментов для решения проблем в самых разных сферах его применения.

Одним из таких инструментов является *Pandas*. *Pandas* – это самая активно развивающаяся библиотека для анализа и обработки данных для языка *Python*. Она позволяет строить сводные таблицы, выполнять группировки, предоставляет удобный доступ к табличным данным, а при наличии рядом другого популярного пакета *matplotlib* дает возможности рисовать графики для полученных групп данных. Релиз *Pandas* был в 2011 году, но теперь она составляет почти 1% запросов в *StackOverflow*.

Существующие для *Python* веб-фреймворки *Django* и *Flask* также набирают популярность. Однако сильного роста популярности не наблюдается. Это говорит о том, что развитие *Python* в настоящее время более связано с анализом данных, а не с веб-разработкой. При этом не стоит забывать, что *Django* используют такие платформы, как *Instagram*, *Mozilla*, *Pinterest* или *Open Stack*.

Ещё одним из наиболее популярных инструментов является *Jupyter Notebook*. Это веб-приложение с открытым исходным кодом, которое позволяет создавать и обмениваться документами, которые содержат живой код, уравнения, визуализации и описательный текст. Использование включает в себя: очистку и преобразование данных, численное моделирование, статистическое моделирование, визуализацию данных, машинное обучение и многое другое. Для построения графиков в *Jupyter* так-

же используется ранее приведенный *matplotlib*. Он поддерживает отображение и редактирование множества форматов данных: изображений, *CSV*, *JSON*, *Markdown*, *PDF*, *Vega*, *Vega-Lite* и прочее. Для быстрой навигации по документам в *Jupyter* есть настраиваемые горячие клавиши, а также возможность использования стандартных сочетаний из *vim*, *emacs* и *Sublime Text*.

Python также популярен и в сфере *DevOps*. *DevOps* – это сочетание культурных принципов, инструментов и практик, которые повышают способность организации поставлять приложения и услуги с высокой скоростью: разрабатывать и улучшать продукты более быстрыми темпами, чем организации, использующие традиционные процессы разработки программного обеспечения и управления инфраструктурой. Это позволяет организациям лучше обслуживать своих клиентов и более эффективно конкурировать на рынке. В *DevOps*, однако, *Python* активно применяется чаще всего без использования как-то дополнительных инструментов для автоматизации развертывания приложений с применением *Docker*. Точно так же стоит учесть, что в веб-разработке *Python* тоже достаточно часто применяется без использования вышеуказанных *Django* и *Flask*.

СФЕРЫ ПРИМЕНЕНИЯ *PYTHON*

На основании популяризации применения *Python* в различных направлениях в работе [4] было рассмотрено то, какие типы компаний чаще всего задают вопросы по *Python* на *StackOverflow*. Исследования были направлены на популярность *Python* в различных сферах стран США и Великобритании.

На рис. 2 приведен график частоты посещения *StackOverflow* по вопросам, связанным с *Python* в различных сферах. Как видно из графика наиболее популярным *Python* является в академических кругах, которые включают колледжи и университеты. Что касается других сфер, то достаточно большого успеха *Python* добился и в сфере правительства, где его использование за последние годы неуклонно растет.

Рост развития *Python* за последние годы достаточно равномерно распространен по отраслям, по крайней мере, в рассмотренных странах США и Великобритании.

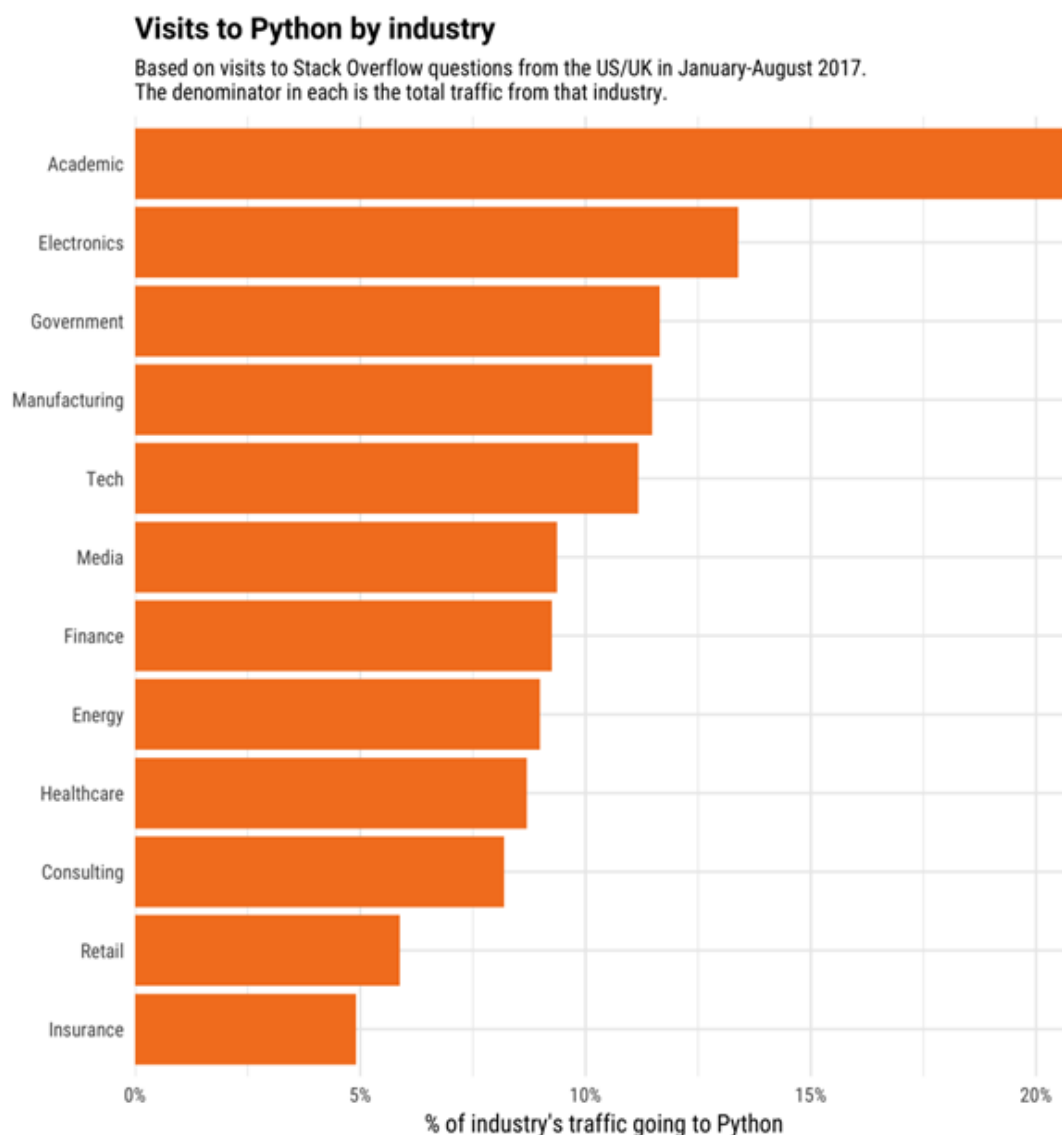


Рис. 2. График частоты посещений *StackOverflow* по вопросам, связанным с *Python* в различных сферах стран США и Великобритании

ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ В *PYTHON*

Как было сказано ранее, поддержка функционального программирования в языке *Python* появилась ещё в самой первой версии и до сих пор пользуется большой популярностью.

Функциональное программирование – это популярная парадигма программирования, приближенная к связи между компьютерной наукой и математикой.

Хоть *Python* и не является полностью функциональным языком, таким как *Haskell*, но он включает некоторые его кон-

цепции наряду с другими парадигмами программирования, используемые в языке. В *Python* легко писать код в функциональном стиле, который может обеспечить лучшее решение для поставленной задачи.

Функциональные языки программирования являются декларативными, они сообщают компьютеру, какой результат они хотят получить, в отличие от императивных языков, которые сообщают компьютеру, какие шаги надо предпринять для решения проблемы. *Python* объединяет эти два стиля и, обычно, использует парадигму императивного подхода, но при необходимости может использовать декларативный стиль.

На некоторые особенности *Python* повлиял *Haskell*, который, как было сказано ранее, является чисто функциональным языком программирования. Чтобы лучше дать понятие функционального языка далее приведены функции *Haskell*, которые можно рассматривать как желательные черты функционального языка программирования:

- чистые функции не должны иметь побочных эффектов, то есть они не меняют состояние программы. При одинаковом входе чистая функция всегда будет выдавать один и тот же результат. «Чистой» называется функция, которая детерминирована и не имеет побочных эффектов. В этом смысле чистые функции близки к математическим. Они просты для чтения и отладки. Так как они не зависят от чего-то, то порядок их вызова не имеет значения;

- неизменяемость – данные не могут быть изменены после того, как они созданы. Неизменяемость также называют иммутабельностью (*immutable*) и обозначают объекты, которых не могут изменять состояние, а результатом каждой модификации такого объекта будет новый объект, при этом старый изменяться не будет;

- функции высшего порядка – это такие функции, которые могут принимать другие функции в качестве параметров, также они могут возвращать новые функции в качестве выходных значений. Это позволяет абстрагироваться от действий, предоставляя гибкость в поведении разрабатываемого кода.

Haskell также оказал влияние на итераторы и генераторы в *Python*, посредством использования «ленивой» загрузки, однако эта функциональность не является необходимым свойством функционального языка программирования.

Преимущества функционального подхода заключаются в следующем:

- *модульность*. Написание кода в функциональном стиле требует в определенной степени разделения участков программы. Такой код проще использовать, так как такая функция не зависит от внешних переменных или состояний объекта, поэтому не составляет труда вызывать его из других участков кода;

- *краткость*. Функциональное программирование является менее многословным, чем другие стили;

- *параллельность*. Чисто функциональные функции являются поточно-ориентированными и могут беспрепятственно работать в нескольких потоках за счет атомарности данных;

- *тестируемость*. Тестирование функциональной программы невероятно простое. Все что нужно для тестирования такой программы – это набор входных и ожидаемый набор выходных данных. Такие функции являются идемпотентными, то есть повторный вызов одной и той же функции с одним и тем же набором аргументов всегда будет возвращать один и тот же результат.

Python без дополнительных библиотек предоставляет инструменты для написания кода в функциональном стиле. Для создания «чистых» функций достаточно написать функцию так, чтобы, например, при изменении списка, получаемого в качестве аргументов, она возвращала его измененную копию.

Неизменяемость в списках реализована в *Python* отдельной структурой данных, называемой кортежами (*tuple*). По сути, это просто неизменяемый список, хотя кортежи имеют ряд других преимуществ:

- защищены от изменений, как от намеренных, так и от случайных;

- используют меньше памяти в отличие от списков;

- могут быть использованы в качестве ключей для словаря;

– все операции над списками, не изменяющие список можно применять и для работы с кортежами.

Применение функций высшего порядка так же уже реализовано в *Python*. Для этого достаточно просто передать в функцию в качестве аргумента название функции без использования круглых скобок.

Неотъемлемой частью функционального программирования также можно считать лямбда-выражения. Лямбда-выражения еще называют анонимными функциями. В основном, они предназначены для создания небольших анонимных функций, которые используются, например, в качестве аргументов для некоторых встроенных в *Python* функций.

В *Python* существует ряд функций высшего порядка, которые работают непосредственно с лямбда-выражениями и могут достаточно легко обрабатывать такие итерируемые объекты, как списки и итераторы. Для эффективного потребления памяти они возвращают итератор на объект. К этим функция относятся *map*, *filter*, *reduce*.

Популярной функцией *Python*, которая заметно выделяется из остальных функциональных языков программирования, являются генераторные выражения (*list comprehensions*).

Стоит отметить, что несмотря на то, что *Python* поддерживает декларативную и императивную парадигму программирования, более приемлемо вести разработку только в одном стиле. А так как *Python* не имеет поддержки всех возможностей функциональных языков программирования, сообщество *Python* рекомендует использовать императивную парадигму программирования в своих проектах. Тем не менее использование преимуществ функционального программирования во многих случаях сильно упрощает и улучшает качество написанного кода. Однако следует помнить, что код в первую очередь должен быть понятен другому разработчику, поэтому чрезмерное использование функционального программирования и сложные для понимания «чистые» или анонимные функции не допустимы как в проектах, написанных на языке *Python*, так и на других языках, поддерживающих парадигму функционального программирования.

PEP 8

PEP 8 – это документ, в котором сосредоточены рекомендации по написанию кода на *Python*. Он был написан в 2001 году Гвидо Ван Россумом, Барри Варшавой и Ником Когланом. Основной задачей *PEP 8* является улучшение читабельности кода *Python* и его стандартизации [5].

PEP расшифровывается как *Python Enhancement Proposal*. *PEP* – это документ, который описывает новые функции, предлагаемые для *Python*, и документирует для сообщества такие аспекты языка, как его дизайн и стиль.

Многие начинающие программисты не принимают в расчет соглашения по оформлению кода, что приводит к проблематичному для чтения и понимания коду. Мартин Фаулер в своей книге «Рефакторинг» называет такой код, как «код с запашком». Создатель *Python* говорил, что код читается гораздо чаще, чем пишется. Исходя из этого следует понимать, что никто не обязывает программиста пользоваться соглашениями по оформлению кода, но зачастую, игнорирование данных руководств приводит к созданию сложных для поддержки, модификации и использования программ. Следование *PEP 8* особенно важно, если программист ищет работу по разработке на *Python*. Написание качественного и читаемого кода демонстрирует профессионализм. Это продемонстрирует работодателю, что человек понимает, как правильно структурировать и оформлять свой код.

При работе в команде следование соглашениям по оформлению кода является обязательной практикой. Другим людям, которые, возможно, никогда раньше не видели стиль кодирования, который не соответствует стандартной практике, придется читать и понимать его. Наличие руководств, которые рекомендуется соблюдать, облегчит другим чтение написанного кода.

При написании кода большую часть времени приходится уделять на обдумывание различных наименований: переменных, функций, классов, пакетов и т.д. Выбор разумных наименований поможет понять самому разработчику в будущем или другим разработчикам, читающим код, что представляет собой определенная переменная, функция или класс. Создание по-

дробных наименований, достаточно кратких и, в то же время, достаточно емких, чтобы в полной мере описать участок кода, делает код самодокументируемым.

В табл. 1 приведено соглашение по стилю наименований, указанных в *PEP 8*, некоторые из распространенных соглашений об именах и примеры их использования.

Таблица 1

Соглашение по стилю наименований *PEP 8*

Тип	Соглашение по стилю наименования	Примеры
Функции	Рекомендуется использовать слова в нижнем регистре. Слова разделяются знаком нижнего подчеркивания.	<i>function,</i> <i>my_test_function</i>
Переменные	Рекомендуется использовать слова или буквы в нижнем регистре. Слова разделяются знаком нижнего подчеркивания.	<i>var, my_variable</i>
Классы	Рекомендуется начинать каждое слово с заглавной буквы. Слова не разделяются знаком нижнего подчеркивания. Такой стиль называется верблюжьим.	<i>Model, MyClass</i>
Методы	Рекомендуется использовать буквы и слова в нижнем регистре. Слова разделяются знаком нижнего подчеркивания.	<i>my_method, method</i>
Константы	Рекомендуется использовать заглавные буквы и слова. Слова разделяются знаком нижнего подчеркивания.	<i>CONSTANT,</i> <i>MY_CONSTANT,</i> <i>MY_LONG_CONSTANT</i>
Модули	Рекомендуется использовать короткие слова и буквы в нижнем регистре. Слова разделяются знаком нижнего подчеркивания.	<i>module.py,</i> <i>my_module.py</i>
Пакеты	Рекомендуется использовать короткие слова и буквы в нижнем регистре. Слова между собой не разделяются.	<i>package, mypackage</i>

Но для того, чтобы написать читаемый код, все равно требуется быть осторожным с выбором наименований. В дополнение к выбору правильных стилей имен в коде также требуется тщательно выбирать имена. Однако выбор наименований для переменных, функций, классов и т.д., может оказаться довольно сложной задачей. При написании кода требуется продумать

варианты наименований таким образом, чтобы они предоставляли понимание конкретной структуры. Из этого следует, что лучше использовать описательные наименования, чтобы лучше прояснить, что представляет собой конкретный объект.

Помимо наименований в коде следует соблюдать и остальные правила его стилизации, например, отступы. Вертикальные отступы или пустые строки могут значительно улучшить читаемость кода. Код, написанный без отступов между функциями, классами и прочим является достаточно трудным для чтения. Точно так же, слишком много пустых строк в коде делают его сильно разреженным. Существуют три основных правила использования вертикальных отступов:

- между классами и функциями верхнего уровня рекомендуется оставлять две пустые строки;
- между методами классов и вложенными классами рекомендуется оставлять одну пустую строку;
- рекомендуется использовать пустые строки внутри функций, чтобы показать четкий алгоритм её работы, разделенный по шагам.

Правильное использование вертикальных отступов сильно упростит чтение вашего кода и поможет визуально понимать, на какие разделы разбит код и как они связаны между собой.

Длина строки кода также регламентируется в *PEP 8* и ограничивается 79 символами. Такое ограничение реализовано для того, чтобы имелась возможность открывать несколько файлов на одном экране и избегать переноса строк. Конечно, данное соглашение накладывает некоторые сложности и не всегда имеется возможность его исполнять. Поэтому *PEP 8* вносит ряд дополнительных рекомендаций для оформления переноса строк:

- *Python* допускает перенос строки, если код написан в круглых скобках;
- если код не содержится в круглых скобках, требуется использовать обратный слеш перед переносом на новую строку;
- если разрыв строки происходит вокруг бинарных операторов таких как `+`, `*` и т.д. новая строка должна начинаться с бинарного оператора. Данное правило вытекает из математики,

так как математики считают, что разрыв перед двоичными операторами улучшает читабельность.

В *Python* не применяются фигурные скобки для выделения блоков кода, методов, классов и т.д. Вместо этого используются горизонтальные отступы, использование которых стандартизовано в *PEP 8*. Существуют два основных правила для использования горизонтальных отступов:

- требуется использовать четыре пробела подряд для обозначения отступа;
- пробелы предпочтительнее табуляции.

Стоит отметить, что смешивание пробелов и табуляции в коде *Python* версии 3.x запрещено и приведет к ошибке компиляции, в отличие от версий 2.x. Данная проблема возникает у многих начинающих программистов на *Python*. Поэтому, в случаях, когда при компиляции возникает следующая ошибка «*TabError: inconsistent use of tabs and spaces in indentation*», следует заменить все знаки табуляции на пробельные символы.

В *PEP 8* описаны соглашения по применению комментариев в коде. Важно документировать код, чтобы впоследствии читатели могли его понять. Однако комментарии требуется использовать осторожно, так как они быстро устаревают и содержат не актуальную информацию, что может запутать пользователя вашего кода. В большинстве случаев лучше использовать юнит- и интеграционное- тестирование как основную часть документирования кода. В *PEP* приведены следующие рекомендации по оформлению комментариев в коде:

- длина строки комментариев ограничивается 79 символами;
- используются слова и предложения без сокращений с главными буквами в начале предложений;
- при обновлении кода требуется обновлять и комментарии.

Таким образом, следуя рекомендациям, указанным в *PEP 8*, можно гарантировать чистый и читаемый код. Это полезно самому разработчику, его коллегам и пользователям кода, а также демонстрирует профессиональные навыки перед работодателем.

МОДУЛИ И ПАКЕТЫ *PYTHON*

Модульное программирование – это процесс разбиения большой громоздкой задачи на отдельные, более мелкие, более управляемые подзадачи или модули. Впоследствии отдельные модули можно объединить, как базовые компоненты, для создания более крупного приложения.

Существует ряд преимуществ применения модульного программирования:

- *Простота*. Вместо того, чтобы сосредоточиться на всей проблеме, модуль обычно фокусируется на одной относительно небольшой части проблемы. При работе с одним модулем имеется гораздо меньшее количество возможных проблем. Это делает разработку проще и менее подверженной ошибкам и так же упрощает отладку и тестирование кода.

- *Модифицируемость*. Модули обычно разрабатываются таким образом, чтобы обеспечить соблюдение логических границ между различными областями программы. Если модули написаны таким образом, чтобы свести к минимуму взаимозависимость, снижается вероятность того, что модификации одного модуля окажут влияние на другие части программы. Это делает программу более устойчивой к изменениям.

- *Возможность повторного использования*. Функциональность, определенная в одном модуле, может быть легко использована повторно другими частями приложения. Это устраняет необходимость дублирования кода.

- *Область действия*. Модули обычно определяют отдельное пространство имен, что помогает избежать коллизий между идентификаторами в разных областях программы. Это один из принципов Дзен питона – «Пространства имен» – одна из замечательных идей – давайте сделаем больше!

В *Python* существует три способа определения модулей:

- модуль может быть написан на *Python*;
- модуль может быть написан на *C* и загружен динамически во время исполнения;
- встроенный модуль, который поставляется вместе с интерпретатором.

Доступ к содержимому модуля осуществляется одинаково для всех вышеперечисленных способов определения: с помощью оператора *import*.

PYTHON VIRTUAL ENVIRONMENTS

Python, как и большинство других современных языков программирования, имеет свой собственный уникальный способ загрузки, хранения и разрешения пакетов (или модулей). Хотя это приносит определенные преимущества, было принято несколько интересных решений относительно хранения и предоставления доступа к пакетам, что привело к некоторым проблемам, особенно с тем, как и где хранятся пакеты.

Есть несколько разных мест, где внешние пакеты могут быть установлены в системе. Например, большинство системных пакетов хранятся в дочернем каталоге пути, хранящегося в *sys.prefix*. Что касается сторонних пакетов, которые устанавливаются с помощью *pip* или *easy_install*, то они хранятся в одном из каталогов, на которые указывается *site.getsitepackages*.

Это важно знать, потому что по умолчанию каждый проект, находящийся в системе будет использовать эти же каталоги для хранения и извлечения сторонних пакетов. На первый взгляд, это может показаться неважным, и это так, для системных пакетов (пакетов, которые являются частью стандартной библиотеки *Python*), но не для сторонних.

Предположим, существует следующая ситуация, в которой имеется два проекта: *ProjectA* и *ProjectB*, оба из которых зависят от одной и той же библиотеки *ProjectC*. Проблема становится очевидной, когда для обоих проектов требуются разные версии *ProjectC*. Предположим, *ProjectA* требуется версия 1.0.0, в то время как *ProjectB*, например, требуется более новая версия 2.0.0.

Это достаточно болезненная тема для *Python*, поскольку он не может различать версии в каталогах *site.getsitepackage*. Таким образом, и v1.0.0 и v2.0.0 будут находиться в одном и том же каталоге с одинаковым именем.

Для решения данной проблемы существуют виртуальные среды *Python*. По сути, основная их цель – это создание изоли-

рованной среды для проектов *Python*. Это означает, что каждый проект может иметь свои собственные зависимости, независимо от того, какие зависимости есть у каждого другого проекта.

В приведенном ранее примере следует просто создать отдельную виртуальную среду для *ProjectA* и *ProjectB*, чтобы всё работало корректно. Каждая среда, в свою очередь, сможет зависеть от той версии *ProjectC*, которая необходима для конкретного проекта.

Положительным аспектом является тот факт, что не существует ограничений на количество сред, которые можно иметь, поскольку они представляют собой просто каталоги, содержащие несколько скриптов. Кроме того, они легко создаются с помощью инструментов командной строки *virtualenv* или *pyenv*.

Хоть виртуальные среды, безусловно, решают некоторые проблемы с управлением пакетами, они не идеальны. После создания нескольких сред можно заметить, что они создают новые проблемы, большинство из которых вращаются вокруг управления самими средами. Чтобы помочь с этим, был создан инструмент *virtualenvwrapper*. Это всего лишь несколько скриптов-оберток вокруг основного инструмента *virtualenv*.

Virtualenvwrapper имеет ряд полезных функций:

- организует все виртуальные среды в одном месте;
- предоставляет методы, которые помогут легко создавать, удалять и копировать среды;
- предоставляет одну команду для переключения между средами.

Некоторые из этих функций могут показаться незначительными, но при работе с виртуальным окружением они являются достаточно важными инструментами, направленными на повышение качества и скорости рабочего процесса.

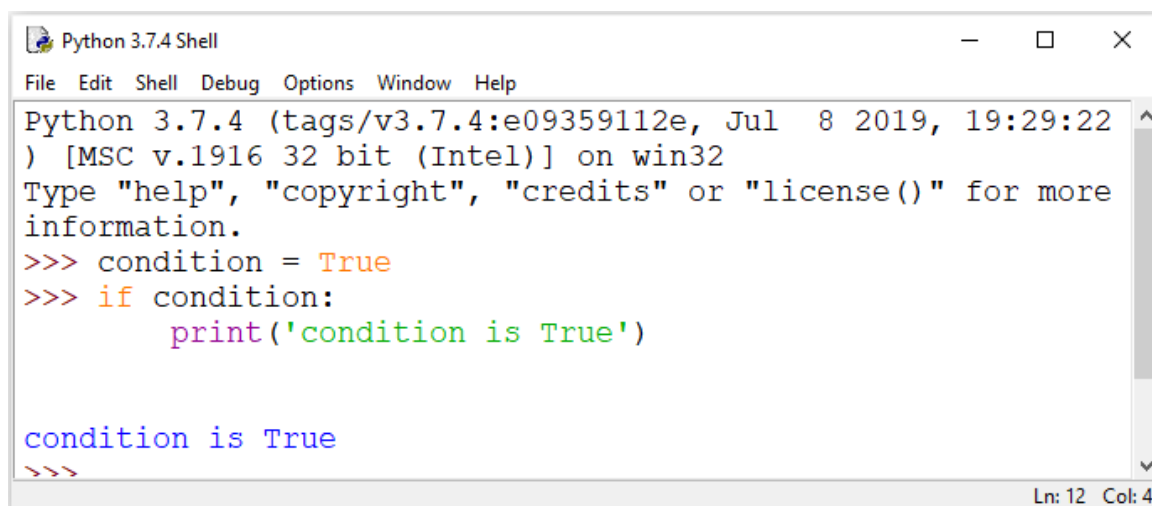
РАБОТА В IDLE. ИСПОЛЬЗОВАНИЕ АРИФМЕТИЧЕСКИХ ОПЕРАЦИЙ. СОЗДАНИЕ СПИСКОВ И СЛОВАРЕЙ. РАБОТА С ЦИКЛАМИ

Цель работы: научиться использовать *IDLE* при работе с *Python*; изучить списки и словари, освоить основные методы для работы с ними; изучить виды циклов в *Python*.

Краткая теория

Среда разработки *IDLE*. После установки *Python 3* автоматически устанавливается *IDLE* – это интегрированная среда разработки, которая включает подсветку синтаксиса, отладчик, *Python Shell* и полноценную документацию по *Python 3* [6-8].

При запуске *IDLE* открывается окно командной оболочки *Python* – *REPL*-среда. Она запрашивает инструкцию от пользователя отображая подсказку в виде трех знаков больше (`>>>`). При использовании данной среды после ввода блока кода он незамедлительно выполняется и отображается результат его работы. Пример написания кода в *IDLE* в окне командной оболочки *Python* приведен на рис. 1.1.



```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul 8 2019, 19:29:22) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> condition = True
>>> if condition:
    print('condition is True')

condition is True
>>>
```

Рис. 1.1. Пример написания кода в *IDLE* в окне командной оболочки *Python*

IDLE понимает синтаксис *Python* и предлагает подсказки по завершению кода, когда используются встроенные функции типа *print()*. Функция *print()* выводит сообщение на стандарт-

ное устройство вывода (обычно экран). Стоит отметить, что в отличие от C-подобных языков программирования вместо фигурных скобок (`{}`), определяющих границы блока кода, в *Python* используются отступы. При этом стоит отметить, что *Python 3* запрещает смешивание табуляции и пробелов в отступах. Из рис. 1.1 видно, что переменная в *Python* состоит из наименования, знака присваивания и значения.

Однако полноценно писать программы в окне командной оболочки *Python* не удобно, он более подходит для тестирования какого-либо небольшого блока кода. Для написания полноценных скриптов используется окно текстового редактора.

Для создания файла *Python* требуется последовательно выбрать пункты меню *File -> New File*, после чего откроется окно редактирования. В данном окне можно писать полноценную программу. Для того, чтобы запустить созданный скрипт, требуется последовательно выбрать пункты меню *Run -> Run Module* или нажать клавишу *F5*. При этом *IDLE* попросит сохранить созданный файл, после чего результат выполнения написанного кода отобразится в окне командной оболочки *Python*.

На рис. 1.2 приведен пример выполнения кода в *IDLE* с помощью окна редактора.

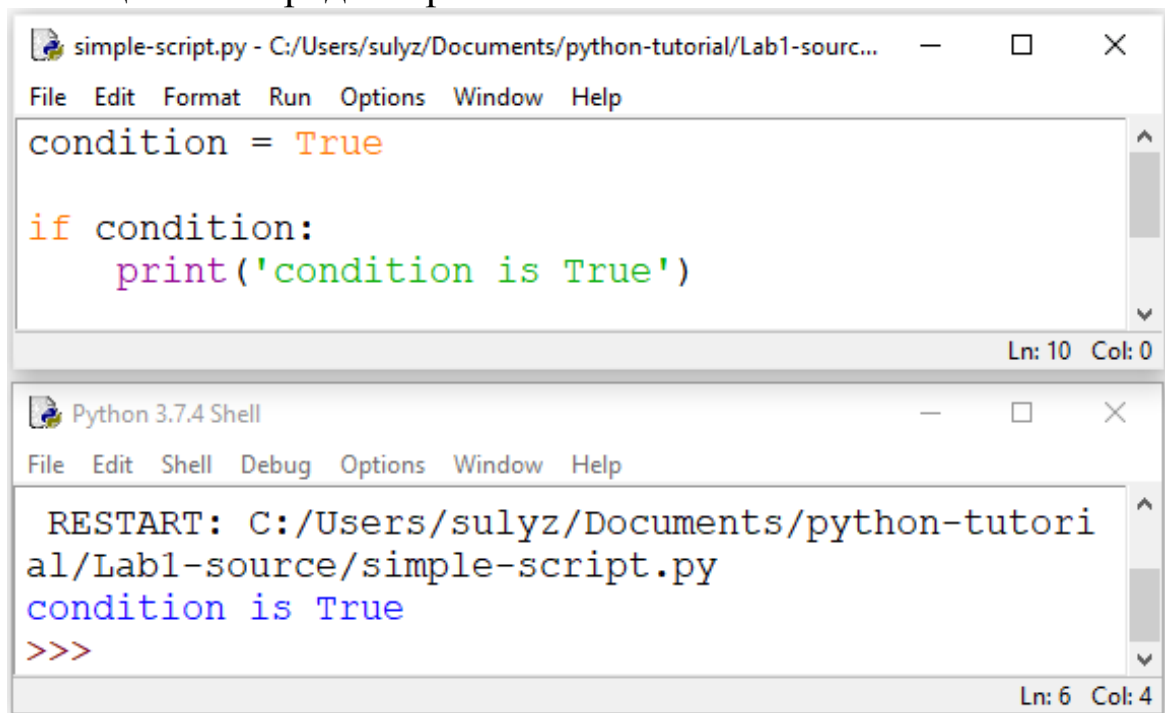


Рис. 1.2. Пример выполнения кода в *IDLE* с помощью окна редактора

IDLE имеет множество инструментов для работы с кодом. Рассмотрим некоторые из них:

- чтобы *IDLE* предложил варианты завершения встроенной функции, требуется нажать на кнопку *TAB*, после чего выбрать необходимую функцию из списка;

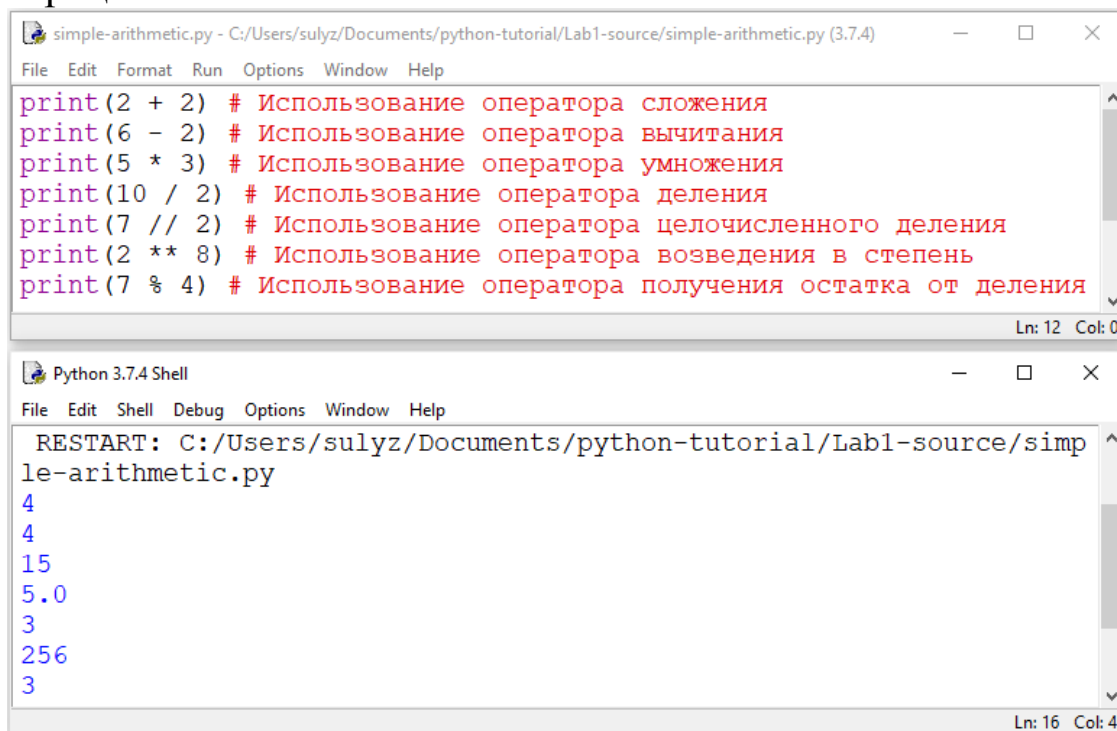
- в окне командной оболочки *Python* используются комбинации клавиш *ALT+P* для вызова предыдущего блока кода и *ALT+N* – для вызова следующего.

Таким образом, можно отметить, что *IDLE* является неплохим решением для написания кода на *Python*. Однако это не самая лучшая *IDE*, но её, как минимум, хватит для выполнения учебных задач.

Арифметические операции

В *Python*, как и в других языках программирования существуют арифметические операторы сложения, вычитания, умножения и деления. Отдельно стоит выделить операторы целочисленного деления двух чисел (*//*), оператор возведения в степень (****) и оператор получения остатка от деления (*%*).

На рис. 1.3 приведен пример применения арифметических операций.



The screenshot displays the Python IDLE interface. The top window, titled 'simple-arithmetic.py - C:/Users/sulyz/Documents/python-tutorial/Lab1-source/simple-arithmetic.py (3.7.4)', contains the following Python code:

```
print(2 + 2) # Использование оператора сложения
print(6 - 2) # Использование оператора вычитания
print(5 * 3) # Использование оператора умножения
print(10 / 2) # Использование оператора деления
print(7 // 2) # Использование оператора целочисленного деления
print(2 ** 8) # Использование оператора возведения в степень
print(7 % 4) # Использование оператора получения остатка от деления
```

The bottom window, titled 'Python 3.7.4 Shell', shows the output of the script after execution:

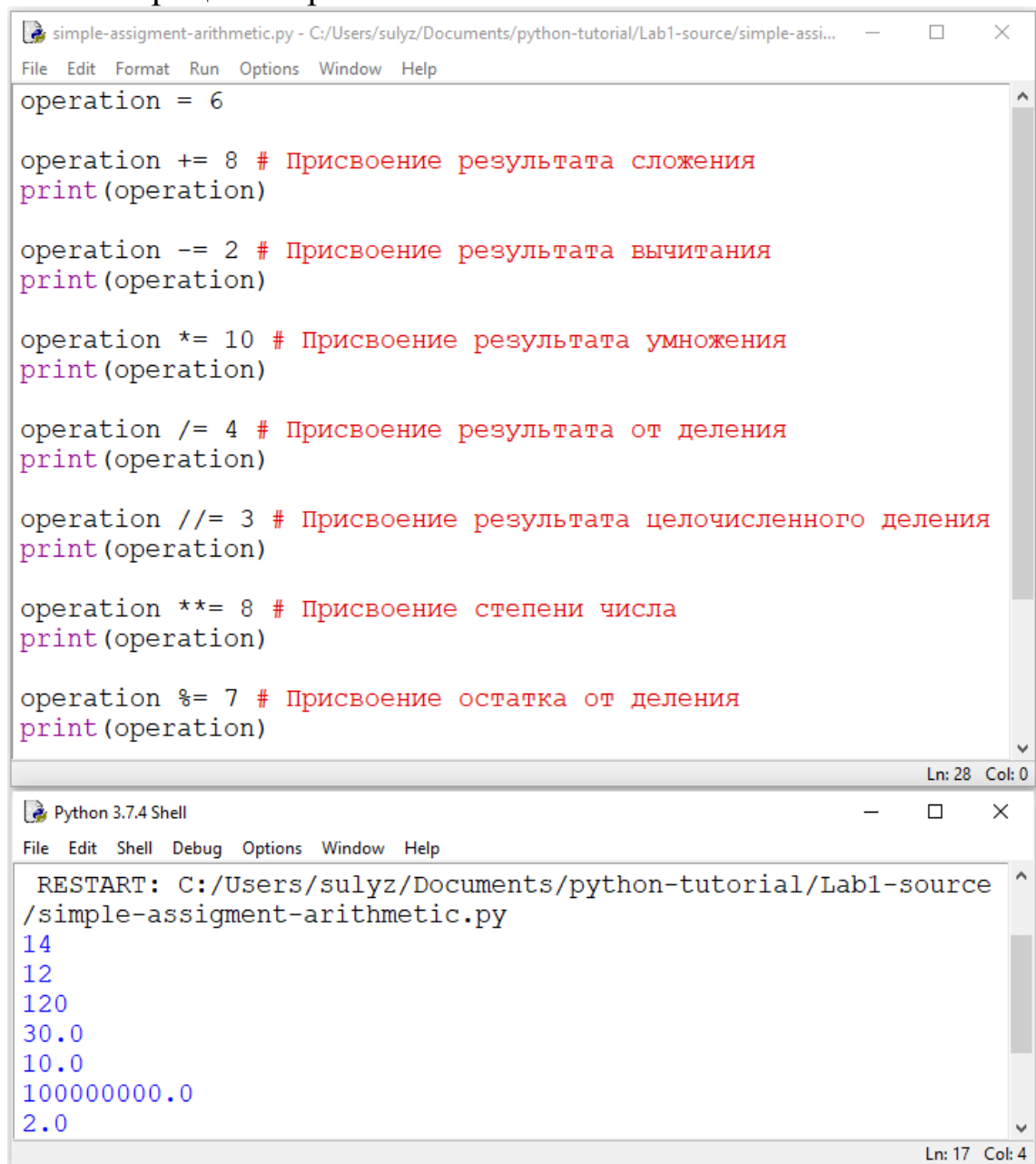
```
RESTART: C:/Users/sulyz/Documents/python-tutorial/Lab1-source/simp
le-arithmetic.py
4
4
15
5.0
3
256
3
```

Рис. 1.3. Пример применения арифметических операций

Последовательное применение арифметических операторов производится исходя из их приоритета (первый знак имеет наибольший приоритет): «*», «/», «%», «+», «-». Как и в других языках программирования, приоритетную арифметическую операцию можно вынести в круглые скобки.

В *Python* используются арифметические операции с присвоением. Они состоят из арифметического оператора и знака присваивания (=). Например, «+=» представляет собой присвоение результата сложения.

На рис. 1.4 приведен пример использования арифметических операций с присвоением.



The image shows a screenshot of a Python IDE. The top window, titled 'simple-assignment-arithmetic.py', contains the following code:

```
operation = 6

operation += 8 # Присвоение результата сложения
print(operation)

operation -= 2 # Присвоение результата вычитания
print(operation)

operation *= 10 # Присвоение результата умножения
print(operation)

operation /= 4 # Присвоение результата от деления
print(operation)

operation //= 3 # Присвоение результата целочисленного деления
print(operation)

operation **= 8 # Присвоение степени числа
print(operation)

operation %= 7 # Присвоение остатка от деления
print(operation)
```

The bottom window, titled 'Python 3.7.4 Shell', shows the output of the script after a restart:

```
RESTART: C:/Users/sulyz/Documents/python-tutorial/Lab1-source
/simple-assignment-arithmetic.py
14
12
120
30.0
10.0
100000000.0
2.0
```

Рис. 1.4. Пример использования арифметических операций с присвоением

Списки и словари

Перед написанием кода следует рассмотреть две важные встроенные структуры данных *Python* – списки и словари [9]. Список представляет собой упорядоченную изменяемую коллекцию объектов произвольных типов, что отличает его от привычных массивов. Списки, как и массивы в других языках программирования, заключаются в квадратные скобки (`[]`). Для объявления пустого массива необходимо присвоить переменной квадратные скобки, а для создания списка с данными необходимо добавить в квадратные скобки значения через запятую. Значениями могут быть любые объекты, в том числе и другие списки. Следует выделить, что с помощью встроенной функции `list()` к списку можно привести любой итерируемый объект.

Выделяют следующие основные методы для работы со списками:

- `append(e)` – добавляет элемент *e* в конец списка;
- `extend(L)` – расширяет список, для которого вызывается, добавляя в конец все элементы списка *L*;
- `insert(i, e)` – вставляет элемент *e* на позицию *i*;
- `remove(e)` – удаляет первый найденный элемент *e*;
- `count(e)` – возвращает количество элементов *e* в списке;
- `clear()` – полностью очищает список.

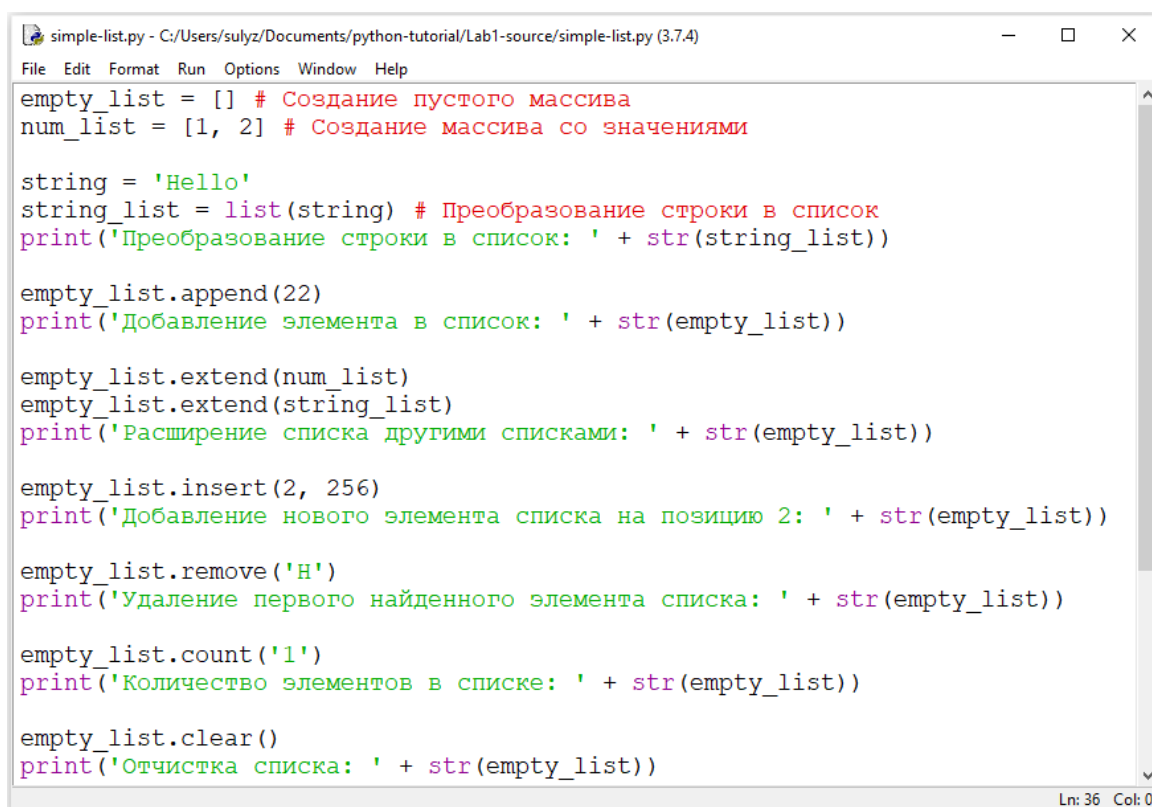
На рис. 1.5 приведен пример работы со списками и использования вышеприведенных методов. На рис. 1.6 приведен результат работы с методами списков.

Второй важной структурой данных в *Python* являются словари. Словарь – это неупорядоченная коллекция объектов с типом хранения «ключ-значение». Для объявления пустого словаря используются фигурные скобки (`{}`), а для объявления словаря со значениями необходимо указать ключ и после знака двоеточия указать значение, ключ со значением далее указывается через запятую.

Выделяют следующие основные методы работы со словарями:

- `get(key)` – возвращает значение ключа;
- `items()` – возвращает пары (ключ, значение);

- *keys()* – возвращает ключи в словаре;
- *values()* – возвращает значения в словаре;
- *update(d)* – добавляет в словарь пары (ключ, значение);
- *pop(key)* – удаляет ключ *key* и возвращает значение.



```

simple-list.py - C:/Users/sulyz/Documents/python-tutorial/Lab1-source/simple-list.py (3.7.4)
File Edit Format Run Options Window Help
empty_list = [] # Создание пустого массива
num_list = [1, 2] # Создание массива со значениями

string = 'Hello'
string_list = list(string) # Преобразование строки в список
print('Преобразование строки в список: ' + str(string_list))

empty_list.append(22)
print('Добавление элемента в список: ' + str(empty_list))

empty_list.extend(num_list)
empty_list.extend(string_list)
print('Расширение списка другими списками: ' + str(empty_list))

empty_list.insert(2, 256)
print('Добавление нового элемента списка на позицию 2: ' + str(empty_list))

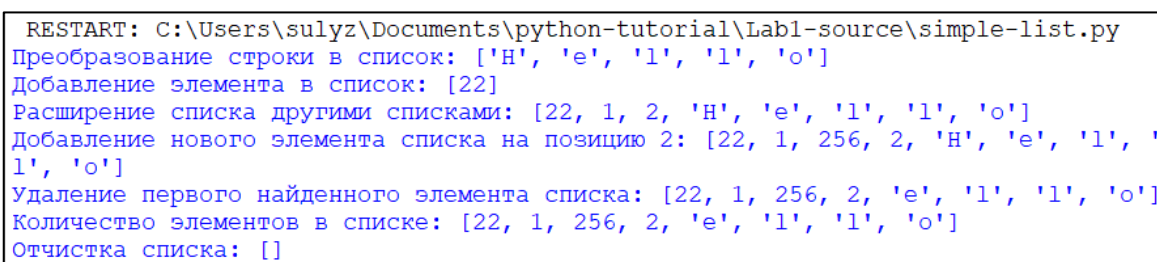
empty_list.remove('H')
print('Удаление первого найденного элемента списка: ' + str(empty_list))

empty_list.count('l')
print('Количество элементов в списке: ' + str(empty_list))

empty_list.clear()
print('Отчистка списка: ' + str(empty_list))
Ln: 36 Col: 0

```

Рис. 1.5. Пример работы со списками и использования их основных методов



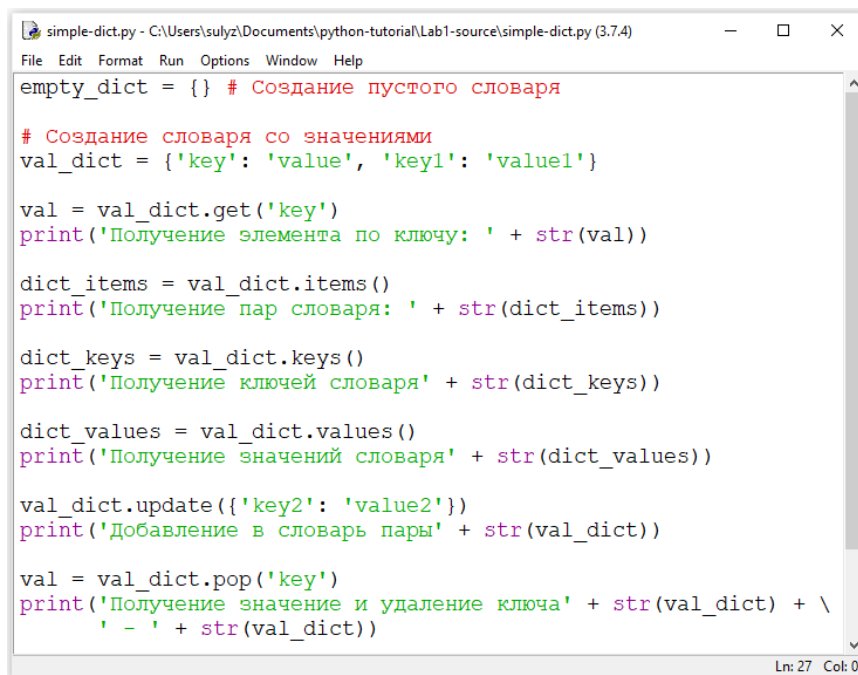
```

RESTART: C:\Users\sulyz\Documents\python-tutorial\Lab1-source\simple-list.py
Преобразование строки в список: ['H', 'e', 'l', 'l', 'o']
Добавление элемента в список: [22]
Расширение списка другими списками: [22, 1, 2, 'H', 'e', 'l', 'l', 'o']
Добавление нового элемента списка на позицию 2: [22, 1, 256, 2, 'H', 'e', 'l', 'l', 'o']
Удаление первого найденного элемента списка: [22, 1, 256, 2, 'e', 'l', 'l', 'o']
Количество элементов в списке: [22, 1, 256, 2, 'e', 'l', 'l', 'o']
Отчистка списка: []

```

Рис. 1.6. Результат работы методов списков

На рис. 1.7 приведен пример работы со словарями и использования вышеприведённых методов. На рис. 1.8 приведен результат работы с методами словарей.



```
simple-dict.py - C:\Users\sulyz\Documents\python-tutorial\Lab1-source\simple-dict.py (3.7.4)
File Edit Format Run Options Window Help

empty_dict = {} # Создание пустого словаря

# Создание словаря со значениями
val_dict = {'key': 'value', 'key1': 'value1'}

val = val_dict.get('key')
print('Получение элемента по ключу: ' + str(val))

dict_items = val_dict.items()
print('Получение пар словаря: ' + str(dict_items))

dict_keys = val_dict.keys()
print('Получение ключей словаря' + str(dict_keys))

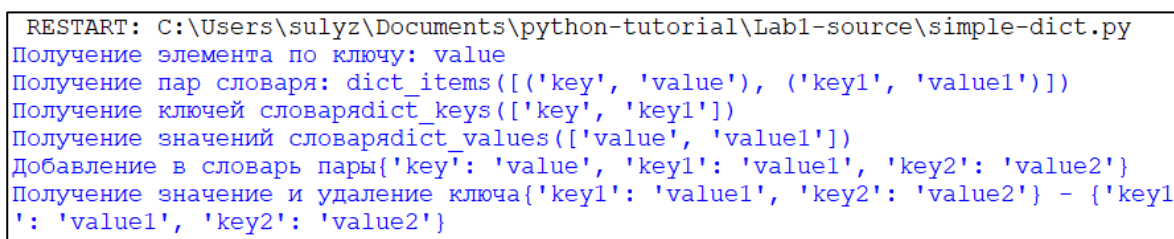
dict_values = val_dict.values()
print('Получение значений словаря' + str(dict_values))

val_dict.update({'key2': 'value2'})
print('Добавление в словарь пары' + str(val_dict))

val = val_dict.pop('key')
print('Получение значение и удаление ключа' + str(val_dict) + \
      ' - ' + str(val_dict))

Ln: 27 Col: 0
```

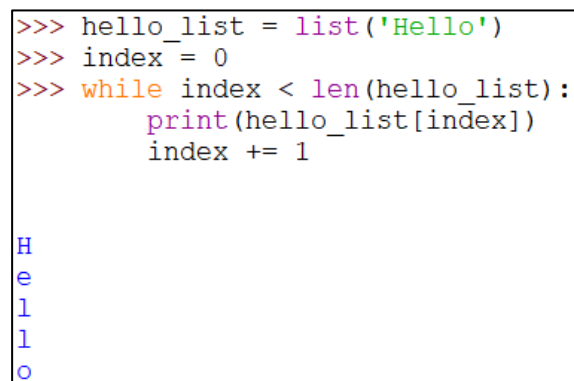
Рис. 1.7. Пример работы со словарями и использования их основных методов



```
RESTART: C:\Users\sulyz\Documents\python-tutorial\Lab1-source\simple-dict.py
Получение элемента по ключу: value
Получение пар словаря: dict_items([('key', 'value'), ('key1', 'value1')])
Получение ключей словаряdict_keys(['key', 'key1'])
Получение значений словаряdict_values(['value', 'value1'])
Добавление в словарь пары{'key': 'value', 'key1': 'value1', 'key2': 'value2'}
Получение значение и удаление ключа{'key1': 'value1', 'key2': 'value2'} - {'key1': 'value1', 'key2': 'value2'}
```

Рис. 1.8. Результат использования основных методов словарей

Для полноценной работы со словарями и массивами требуется рассмотреть циклы. Их в *Python* всего два вида: *for* и *while*. Цикл *while* аналогичен циклу в *C*-подобных языках. Он выполняется до тех пор, пока указанное в нем условие истинно. На рис. 1.9 приведено использование цикла *while* для вывода всех элементов списка.



```
>>> hello_list = list('Hello')
>>> index = 0
>>> while index < len(hello_list):
    print(hello_list[index])
    index += 1

H
e
l
l
o
```

Рис. 1.9. Пример использования цикла *while* для вывода всех элементов списка

Как видно из рис. 1.9, для получения значения количества элементов списка вызывается функция *len()*.

Цикл *for* в *Python* отличается по виду от *C*-подобных языков и используется для прохода по итерируемым объектам. Данный цикл похож на цикл *foreach* в некоторых других языках программирования. На рис. 1.10 приведен пример использования цикла *for* для вывода всех элементов списка, а на рис. 1.11 – для вывода всех элементов словаря.

```
>>> int_list = [1, 2, 3]
>>> for el in int_list:
    print(el)

1
2
3
```

Рис. 1.10. Пример использования цикла *for* для вывода всех элементов списка

```
>>> city_university_dict = {'Novoch': ['NPI', 'NIMI'], 'Rostov': ['DGTU', 'YFU']}
>>> for city, university in city_university_dict.items():
    print('City ' + str(city) + ' has ' + str(len(university)) + ' univers')

City Novoch has 2 univers
City Rostov has 2 univers
```

Рис. 1.11. Пример использования цикла *for* для вывода всех элементов словаря

Варианты заданий

1. Написать функцию, которая принимает целочисленный список с количеством элементов 1 или более и возвращает *True*, если цифра 6 является первым или последним элементом списка.

2. Написать функцию, которая принимает два целочисленных списка и возвращает *True*, если первые или последние элементы данных списков равны. Оба списка содержат 1 или более элементов.

3. Написать функцию, которая принимает целочисленный список, состоящий из трех элементов, и возвращает новый перевернутый список. Например, на входе {1,2,3}, а на выходе {3,2,1}.

4. Написать функцию, которая принимает два целочисленных списка, содержащих по три элемента каждый, и возвращает новый список, состоящий из двух элементов, являющихся средними во входящих списках. Например, для следующих входящих списков $\{1,2,3\}$ и $\{3,4,5\}$ итоговым будет $\{2,4\}$.

5. Написать функцию, которая принимает целочисленный список и возвращает *True*, если длина списка больше нуля и первый и последний элемент списка равны.

6. Написать функцию, которая принимает целочисленный список, содержащий три элемента, и возвращает сумму этих элементов.

7. Написать функцию, которая принимает целочисленный список, содержащий три элемента, и изменяет данный список путем изменения всех его элементов на максимальный крайний элемент списка. Например, для входящего списка $\{1,2,3\}$ изменённым будет $\{3,3,3\}$.

8. Написать функцию, которая принимает целочисленный список, состоящий из n элементов, и возвращает *True*, если он содержит числа 2 или 3.

9. Написать функцию, которая принимает целочисленный список, состоящий из n элементов, и возвращает количество четных чисел в списке.

10. Написать функцию, которая принимает целочисленный список, состоящий из n элементов, и возвращает сумму элементов списка. Однако стоит исключить из подсчета число 13 и числа, которые следуют после него. Например, для входящего списка $\{1,2,3,13,4\}$ сумма будет равна 6.

11. Написать функцию, которая принимает целочисленный список, состоящий из n элементов, и возвращает *True*, если в каком-то месте списка по порядку содержатся элементы 1,2,3 или комбинации их перестановок.

12. Написать функцию, которая принимает два словаря и возвращает новый словарь, в котором значения одинаковых ключей из входящих словарей являются суммой, а разные ключи

чи просто добавляются в новый словарь. Например, для входящих словарей $\{ 'a': 200, 'b': 50 \}$ и $\{ 'a': 100, 'c': 500 \}$ выходным словарем будет $\{ 'a': 300, 'b': 50, 'c': 500 \}$.

13. Написать функцию, которая принимает два списка и возвращает словарь, в котором ключами выступают элементы первого списка, а значениями ключей – элементы второго. Требуется предусмотреть ситуации, когда один из списков или оба будут пустыми. В ситуации, когда один список по длине больше другого, последние элементы большего по длине списка, не учитывать.

14. Написать функцию, которая принимает список, состоящий из n словарей, где в каждом словаре присутствуют ключи *status* (*True* или *False*) и *name* (строка). Вернуть список, который будет содержать все значения ключей *name* в тех словарях, где значение *status* является *True*. В случае отсутствия таковых вернуть пустой список.

15. Написать функцию, которая принимает на вход список, состоящий из n различных элементов. Вернуть словарь, в котором ключами являются элементы входящего списка, а их значениями – число повторений этих элементов во входящем списке.

Лабораторная работа №2

РАБОТА С ОСНОВНЫМИ ВСТРОЕННЫМИ ФУНКЦИЯМИ

Цель работы: рассмотреть основные встроенные функции языка программирования *Python* и научиться с ними работать.

Краткая теория

Стандартная библиотека *Python* предоставляет огромное количество различных модулей и функций, так называемых *Built-in Functions*. Все эти инструменты поставляются вместе с интерпретатором и их достаточно для того, чтобы немедленно начать работу с языком сразу после установки. Рассмотрим некоторые из основных встроенных функций:

print(*objects, sep='', end='\n', file=sys.stdout, flush=False)

Данная функция предназначена, как было описано в лабораторной работе №1, для вывода сообщения на стандартное устройство вывода. Однако стоит дополнительно рассмотреть её параметры:

- **objects* – параметр, который принимает объекты для вывода на устройства вывода. Символ «*» говорит о том, что параметр принимает переменное количество аргументов, например, строки через запятую;

- *sep* – параметр, аргумент которого предназначен для разделения аргументов параметра **objects*;

- *end* – параметр, аргумент которого выводится в конце вывода аргументов параметра **objects*;

- *file* – параметр, принимающий в качестве аргумента только объекты с методом *write(string)*, если он не указывается или указан как *None* будет использоваться *sys.stdout*;

- *flush* – параметр, который предназначен для дополнительного указания функции при использовании параметра *file* в случае, если он *Истина*, то производить немедленный сброс данных из буфера в память.

Дополнительно стоит отметить, что оба параметра *sep* и *end* должны принимать только строки. Они могут принимать

None, что иницирует использование значений по умолчанию. Также стоит учесть, что если **objects* не будет принимать никаких аргументов, то выведется только значение *end*.

len(s)

Данная функция принимает в качестве аргумента последовательность (строка, байты, список, кортежи и другое) или коллекции (словарь, множество и другие). Возвращает длину (число элементов) объекта.

int([x])

Возвращает целочисленный объект из числа или строки, переданной в параметр *x*. В случае, если аргументы не заданы, то возвращает 0. У вещественных чисел удаляется дробная часть.

float([x])

Возвращает число с плавающей точкой из числа или строки, переданной в параметр *x*. В случае, когда аргументом параметра *x* является строка, то она должна содержать десятичное число с необязательным предшествующим знаком (+ или -). Если аргумент не задан, то возвращается 0.0.

list([iterable])

Функция предназначена для конструирования списка. Аргумент параметра *iterable* может быть последовательностью, контейнером, либо итераторным объектом. Если в качестве аргумента *iterable* передается уже существующий список, то функция возвращает копию объекта переданного списка. Если заданных аргументов нет, то возвращается новый пустой список.

dict(**kwarg)

Функция предназначена для создания словаря. Два символа «*» означают, что параметр принимает переменное значение именованных аргументов (*one=1, two=2*). В случае, если аргумент передан параметру ***kwarg*, то будет создаваться новый словарь с теми же парами ключ-значение. Если ключ в аргументе присутствует более одного раза, то будет использоваться последнее значение ключа переданного аргумента. Если аргумент не задан, то функция вернет новый пустой словарь.

range(start, stop[, step])

Функция предназначена для генерации неизменяемой числовой последовательности. Данную функцию можно вызывать тремя способами:

— *range(stop)*. В этом случае вернется ряд чисел, начинающихся с 0 и включающих каждое число до аргумента параметра *stop*, но не включающее само значение аргумента;

— *range(start, stop)*. В этом случае вернется ряд чисел, начинающихся со значения аргумента параметра *stop* и включающих каждое число до значения параметра *start*, но не включающее само значение его аргумента;

— *range(start, stop, step)*. В этом случае функция будет работать так же, как и в предыдущем случае, за исключением того, что аргумент параметра *step* будет указывать на разницу между двумя соседними числами.

Следует отметить, что данная функция создает не список, а отдельный тип неизменяемой последовательности.

sum(iterable[, start])

Функция возвращает сумму значений аргумента параметра *iterable*, начиная со *start*. По умолчанию параметр *start* равен 0.

С большим количеством встроенных функций можно ознакомиться в документации по *Python* в разделе «*The Python Standard Library – Built-in Functions*» или по ссылке [10].

Варианты заданий

1. Написать функцию, которая принимает целое число и возвращает сгенерированный список, содержащий количество элементов от 0 до входящего числа включительно.

2. Написать функцию, которая принимает список, который содержит строки и выводит на экран новые списки из входящих строк.

3. Написать функцию, которая принимает список и выводит в консоль значения списка через точку с запятой. Требуется

решить задачу одной строкой с использованием списка с оператором «*».

4. Написать функцию, которая принимает список, состоящий из n элементов, и возвращает их сумму.

5. Написать функцию, которая принимает список, состоящий из строк, которые являются целочисленными значениями или значениями с плавающей точкой, и возвращает их сумму.

6. Написать функцию, которая принимает список, состоящий из n элементов, и возвращает количество целых чисел и чисел с плавающей точкой.

7. Написать функцию, которая принимает целочисленный список, состоящий из n элементов, и записывает в файл новый список, состоящий из тех элементов входящего списка, значения квадрата которых не превышают 30.

8. Написать функцию, которая принимает целочисленный список, состоящий из n элементов, и возвращает список, состоящий из новых списков, элементами которых являются числа от 0 до значений элемента во входящем списке включительно. Например, входящий список имеет вид [1,2,3] на выходе будет список списков [[0,1], [0,1,2], [0,1,2,3]].

9. Написать функцию, которая принимает целочисленный список, содержащий n элементов, и целое число, и выводит на экран сгенерированные списки, содержащие n элементов с шагом итерации, равном элементу входящего списка.

10. Написать функцию, которая принимает на вход две разных строки и возвращает количество совпадений двух символов в строках. Например, на входе две строки «*xadasw*» и «*xad*» совпадением будет считаться группа символов «*xa*» и «*ad*».

Лабораторная работа №3

**РАБОТА С ИТЕРАТОРАМИ, ГЕНЕРАТОРАМИ.
РАБОТА С ГЕНЕРАТОРНЫМИ ВЫРАЖЕНИЯМИ**

Цель работы: изучить понятия итератора и генератора в *Python*, а также их преимущества; ознакомиться с примерами их пользования.

Краткая теория

Итераторы – популярный поведенческий паттерн проектирования для последовательного обхода коллекции, который позволяет не раскрывать их внутреннего представления.

Итерируемый объект – это такой объект, от которого можно получить итератор. В *Python* итерируемым объектом является такой объект, от которого встроенная функция *iter()* возвращает итератор.

Итератором в Python является объект, который реализует метод `__next__` без аргументов и метод `__iter__`. Метод `__next__` должен вернуть следующий элемент или ошибку *StopIteration*.

Преимущества использования итераторов, как было сказано выше, заключается в возможности «указывать» на определенный объект коллекции и при этом скрывать его структуру. Все последовательности (*list*, *tuple*, *range*) в *Python* являются итерируемыми объектами.

Основным местом работы с итераторами в данной лабораторной работе будет использование цикла *for*. Например, при переборе элементов списка или другой последовательности, используя цикл *for*, фактически происходит обращение к итератору данной последовательности с просьбой выдать следующий элемент. Когда элементы в последовательности заканчиваются, очередное обращение к следующему объекту итератора сгенерируют исключение, которое при использовании цикла *for* обрабатывается незаметно для пользователя. На рис. 3.1 представлен обход списка с помощью цикла *for*, а на рис. 3.2 непосредственное использование итератора в работе со списком.

Для того, чтобы получить итератор последовательности, необходимо в качестве аргумента *x* встроенной функции *iter(x)*

передать эту последовательность. Затем для получения следующего элемента последовательности с помощью другой встроенной функции *next(iter)* в качестве аргумента *iter* передавать полученный итератор.

```
>>> courses = [1, 2, 3, 4]
>>> for course in courses:
>>>     print(str(course) + ', ', end='')

1, 2, 3, 4,
>>>
```

Рис. 3.1. Пример работы итератора через использование цикла *for*

```
>>> courses = [1, 2, 3, 4]
>>> iterator = iter(courses)
>>> print(next(iterator))
1
>>> print(next(iterator))
2
>>> print(next(iterator))
3
>>> print(next(iterator))
4
>>> print(next(iterator))
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    print(next(iterator))
StopIteration
>>>
```

Рис. 3.2. Пример непосредственного использования итератора в работе со списком

Генератор (генераторная функция) – это функция, которая возвращает подвид итератора, генерирующий значения. Основное их отличие в том, что они экономят память за счет того, что хранят не все значения, например, списка, а только его предыдущий элемент, предел и формулу, по которой рассчитывается следующий элемент. Данная функция вместо *return* содержит ключевое слово *yield*, которое возвращает объект-генератор, а

не выполняет сразу весь код. В *Python* имеется возможность создавать генераторный объект в сокращенной форме. Для этого используются круглые скобки [11].

Перебрать генератор можно используя цикл *for* как и при работе с итераторами. Однако стоит заметить, что перебрать второй раз генератор не получится, потому что объект-генератор уже сгенерировал данные по описанной в нем формуле. Поэтому генераторы стоит использовать, когда требуется один раз пройти по итерируемому объекту. На рис. 3.3 приведен пример использования генератора в виде функции, а на рис. 3.4 – в сокращенной форме.

```
>>> from random import random
>>> def generate_random_values(number_val):
>>>     for ind in range(number_val):
>>>         yield random()

>>> generator = generate_random_values(5)
>>> for rand_val in generator:
>>>     print(rand_val)

0.852651495492575
0.8247447699393206
0.9739677776248633
0.17777347084064987
0.3487093434416959
>>> |
```

Рис. 3.3. Пример использования генераторной функции

```
>>> from random import random
>>> generator = (random() for i in range(5))
>>>
>>> for val in generator:
>>>     print(val)

0.18943438221027242
0.4820232853540125
0.2216033089894418
0.8412337666400383
0.29027726525842035
>>>
```

Рис. 3.4. Пример создания генератора в сокращенной форме

В *Python* вводится такое понятие как *list comprehension*, которое в русскоязычном литературном языке встречается в виде генераторных выражений или генераторов списков. В данном пособии будем употреблять первый вариант.

Основная задача использования генераторных выражений – это быстрое создание и заполнение списков, словарей или множеств. Отметим, что генераторные выражения – это всего лишь «синтаксический сахар», иначе говоря, они не вносят никаких дополнительных преимуществ, кроме как удобство. Пример использования генераторных выражений списков и словарей представлен на рис. 3.5.

```
>>> from random import random
>>>
>>> random_list = [random() for ind in range(5)]
>>> print(random_list)
[0.8473688518786204, 0.850083228272142, 0.43566553255732765, 0.08688424971114617, 0.39650142868277394]
>>>
>>> random_dict = {str(ind):random() for ind in range(5)}
>>> print(random_dict)
{'0': 0.8180324845492388, '1': 0.8988074097297781, '2': 0.31461261424494313, '3': 0.4329870289988711, '4': 0.7924225479557185}
>>>
>>> random_if_list = [val for val in random_list if val > 0.5]
>>> print(random_if_list)
[0.8473688518786204, 0.850083228272142]
>>> |
```

Рис. 3.5. Пример использования генераторных выражений

Из рис. 3.5 видно, что для создания списка с помощью генераторного выражения требуется использовать конструкцию, похожую на сокращенную форму генераторной функции за исключением того, что используются квадратные скобки вместо круглых. Для генерации словаря используются фигурные скобки и требуется генерировать пару ключ-значение.

Также в генераторном выражении можно использовать условия. Это отражено с помощью списка *random_if_list* на рис. 3.5. Элемент добавляется в создающийся список только, если соответствует заданному условию.

Таким образом, генераторы являются мощной языковой конструкцией, которая не только упрощает написание кода, но и снижает потребление памяти и мощности ресурсов процессора. Однако не стоит злоупотреблять генераторными выражениями, т.к. в некоторых случаях это может сильно снизить читаемость кода.

Варианты заданий

1. Написать функцию, которая принимает целое число и с помощью генераторного выражения создает и возвращает новый список случайных чисел с длиной входящего числа.

2. Написать функцию, которая принимает три целых числа x , a , b и с помощью генераторного выражения создает и возвращает новый список длиной x случайных чисел от a до b . Для решения данного задания рекомендуется использовать функцию `random.randint()`.

3. Написать функцию, которая принимает целочисленный список, состоящий из n элементов, и с помощью генераторного выражения создает и возвращает список, элементами которого являются удвоенные элементы входящего списка.

4. Написать функцию, которая принимает целочисленный список, состоящий из n элементов, и с помощью генераторного выражения создает и возвращает список, содержащий только четные элементы входящего списка.

5. Написать функцию, которая принимает целочисленный список, состоящий из n элементов, и с помощью генераторного выражения создает и возвращает список, содержащий только положительные элементы входящего списка.

6. Написать функцию, которая принимает словарь и с помощью генераторного выражения создает и возвращает новый список, содержащий значения ключей входящего словаря.

7. Написать функцию, которая принимает два одинаковых по длине списка и с помощью генераторного выражения создает и возвращает новый словарь, в котором ключами являются элементы первого списка, а значениями ключей – элементы второго.

8. Написать функцию, которая принимает список и с помощью генераторного выражения создает и возвращает словарь, где в качестве ключей будут номера позиций элементов входящего словаря, а значениями – сами элементы.

9. Написать функцию, которая принимает целочисленный список, состоящий из n элементов, и с помощью генераторного выражения создает новый массив элементов из тех элементов входящего массива, квадрат числа которых не превышает 30.

10. Написать функцию, которая принимает список, состоящий из n элементов, и целое число x и с помощью генераторной функции выводит на консоль только те элементы входящего списка, которые больше числа x .

11. Написать функцию, которая принимает список, состоящий из n элементов, и два целых числа a и b и с помощью генераторной функции выводит на консоль элементы входящего списка от a до b включительно. Требуется предусмотреть ситуацию, когда значения a или b больше длины списка.

12. Написать функцию, которая принимает список списков и с помощью генераторного выражения создает и возвращает новый список, который содержит все элементы входящих списков.

13. Написать функцию, которая принимает список и с помощью генераторного выражения создает и возвращает новый список, содержащий только уникальные элементы входящего списка.

14. Написать функцию, которая принимает список словарей и с помощью генераторного выражения создает и возвращает новый словарь, который содержит все элементы входящих словарей. Ключи в словарях уникальны.

15. Написать функцию, которая принимает строку и с помощью генераторного выражения создает список, в котором все символы входящей строки смещены на один символ вправо по алфавиту, кроме символов от «ы» до «я». Требуется вернуть строку из созданного списка.

Лабораторная работа №4

РАБОТА С ОСНОВНЫМИ МОДУЛЯМИ

Цель работы: изучить основные модули стандартной библиотеки *Python 3*; рассмотреть модули *os* и *datetime*.

Краткая теория

Модули в *Python* – это файлы с расширением *.py*, которые содержат исполняемый код на *Python*. Именем модуля является имя файла без расширения. Система модулей позволяет логически организовывать код программы. Группирование кода в модули значительно облегчает процесс написания, а что более важно – дальнейшее понимание написанной программы. Для того, чтобы использовать файл *.py* как модуль в другом файле, требуется использовать ключевое слово ***import***.

Стандартная библиотека *Python* содержит достаточно большое количество встроенных модулей. Рассмотрим некоторые из них:

- модуль ***os***. Представляет множество функций для работы с операционной системой. При этом стоит учитывать, что их поведение, как правило, не зависит от конкретной операционной системы, что позволяет разрабатывать кроссплатформенные решения с использованием данного модуля;

- модуль ***datetime***. Содержит классы для работы с датой и временем. В отличие от других языков программирования данный модуль достаточно прост в освоении и использовании;

- модуль ***array***. Представляет собой реализацию массивов на *Python*. Они похожи на списки, но их различает то, что массивы ограничены одним определенным типом данных и размером каждого элемента;

- модуль ***itertools***. Содержит сборник полезных итераторов;

- модуль ***sys***. Модуль, позволяющий обращаться к некоторым переменным и функциям, которые взаимодействуют с интерпретатором *Python*;

- модуль ***random***. Данный модуль, как следует из названия, предназначен для генерации случайной последовательности

чисел, букв и случайного выбора элементов последовательно-сти;

- модуль ***math***. Предоставляет набор функций для выполнения математических операций;

- модуль ***json***. Содержит функции для работы с форматом передачи данных *json*. Предоставляет возможности как парсинга данных, так и создания объектов в данном формате;

- модуль ***gzip*** и ***zlib***. Предоставляет средства для работы со сжатыми файлами;

- модуль ***tkinter***. Данный модуль позволяет создавать кроссплатформенные программы с графическим интерфейсом.

Данные модули всего лишь небольшая часть стандартной библиотеки *Python*. Существуют модули для работы со звуком, базами данных, шифрованием, хешированием, логгированием и многим другим. Остановимся более подробно на некоторых модулях, которые понадобятся в дальнейших работах.

Модуль ***os***

Как было сказано ранее, данный модуль предоставляет функции для работы с операционной системой:

- ***os.listdir(path)*** – функция позволяет получить список файлов и директорий по указанному пути аргумента параметра *path*;

- ***os.mkdir(path)*** – функция позволяет создать новую директорию. В случае, если такая существует, пробрасывает *OSError*;

- ***os.remove(path)*** – функция удаляет файл;

- ***os.rename(src, dst)*** – функция переименовывает файл или директорию из *src* в *dst*;

- ***os.rmdir(path)*** – удаляет пустую директорию.

Стоит выделить также вложенный модуль ***os.path***, который предоставляет средства для работы с путями:

- ***os.path.exists(path)*** – функция возвращает *True*, в случае если *path* указывает на существующий путь или дескриптор открытого файла;

- ***os.path.isfile(path)*** – функция возвращает *True*, если *path* указывает на файл;

- *os.path.isdir(path)* – функция возвращает *True*, если *path* указывает на директорию;
- *os.path.join(path1[, path2[, ...]])* – функция соединяет пути с учетом особенностей операционной системы;
- *os.path.samefile(path1, path2)* – функция возвращает *True*, если *path1* и *path2* указывают на один и тот же файл или директорию.

Модуль *datetime*

- *datetime.date(year, month, day)* – класс *date* создает объект даты по значениям аргументов указанных параметров. В случае, если один из аргументов будет некорректным, сгенерируется исключение *ValueError*;
- *datetime.time(hour=0, minute=0, second=0, microsecond=0, tzinfo=None)* – класс *time* создает объект времени по значениям аргументов указанных параметров;
- *datetime.today()* – возвращает объект *datetime* по текущей дате и времени;
- *datetime.strptime(date_string, format)* – преобразует строку в *datetime*;
- *datetime.strftime(datetime_obj, format)* – формирует строку из объекта *datetime* в указанном формате.

Некоторые из остальных модулей будут рассмотрены в дальнейшем.

Рассмотрим пример программы, которая при старте получает список всех папок в директории, из каждой папки получает файлы, хранящиеся в ней, и создает новую папку, именуя её текущей датой и временем, помещая внутрь пустые файлы с расширением *.txt* и наименованием часов в сутках. Исходный код программы приведен на рис. 4.1, а её вывод в сокращенном виде – на рис. 4.2.

На рис. 4.1 в части импорта представлены варианты импортирования модулей в скрипт. Стоит отметить, что импорт *datetime* происходит из модуля *datetime*. Данное действие может ввести в заблуждение, однако стоит понимать, что модуль *datetime* содержит в себе классы для работы с датой и временем: сам класс *datetime*, *date* и *time*.


```

import os
from datetime import datetime as dt, time
from os import path

base_path = 'C:\\Users\\sulyz\\Documents\\python-tutorial\\Laboratory4\\'

# Выводим список файлов и директорий
dirs = os.listdir(base_path)
for directory in dirs:
    files = os.listdir(path.join(base_path, directory))

    print('-' + str(directory))
    for file in files:
        print('\t -' + str(file))

# Создаем новую директорию и файлы
today_dt_str = dt.strftime(dt.today(), '%Y_%m_%d_%H_%M_%S')
today_dir_path = path.join(base_path, today_dt_str)

os.mkdir(today_dir_path)

# Генерируем список с часами
hours = [time(hour).strftime('%H') for hour in range(24)]

# Создание файлов
for hour in hours:
    open(path.join(today_dir_path, str(hour) + '.txt'), 'a').close()

```

Рис. 4.1. Исходный код программы по рассмотренным модулям

```

-2019_10_26_20_41_33
  -00.txt
  -01.txt
  -02.txt
  -03.txt
  -04.txt
  -05.txt
  -06.txt
  -07.txt
  -08.txt
  -09.txt
  -10.txt
  -11.txt
  -12.txt
  -13.txt
  -14.txt
  -15.txt
  -16.txt
  -17.txt
  -18.txt
  -19.txt
  -20.txt
  -21.txt
  -22.txt
  -23.txt
-2019_10_26_20_41_41
  -00.txt
  -01.txt
  -02.txt
  -03.txt

```

Рис. 4.2. Фрагмент вывода разработанной программы

Следует заметить, что стандартная библиотека – это не единственное место, где можно найти полезные модули. Сообщество *Python* поддерживает обширную коллекцию сторонних модулей. Их можно найти в репозитории сообщества по ссылке <https://pypi.python.org/>. Чтобы установить сторонний модуль, можно воспользоваться системой управления пакетами *pip*, который устанавливается вместе с интерпретатором *Python*. Для установки модуля требуется ввести в окне консоли следующую команду ***pip install <наименование_модуля>***

Варианты заданий

1. Написать функцию, которая принимает объект *datetime* и возвращает *True*, если год полученного значения високосный.
2. Написать функцию, которая принимает строку и возвращает объект *datetime* из этой строки. Формат даты и времени можно использовать любой.
3. Написать функцию, которая принимает целочисленное значение *x* и возвращает текущую дату, прибавив к году *x*.
4. Написать функцию, которая принимает объект *datetime* и возвращает номер недели.
5. Написать функцию, которая принимает объект *datetime* и возвращает номер дня в году.
6. Написать функцию, которая принимает два объекта *datetime* и возвращает число по модулю дней между ними.
7. Написать функцию, которая принимает объект *datetime* и возвращает третью среду месяца.
8. Написать функцию, которая принимает целое число *x* и возвращает список из 20 объектов *datetime* с интервалами между днями равными *x*. За первоначальное значение *datetime* следует принять текущую дату.
9. Написать функцию, которая принимает объект *datetime* и возвращает временную метку (*timestamp*) из данного объекта.

10. Написать функцию, которая принимает строку, содержащую *GMT*, и возвращает смещенное значение текущей даты и времени.

11. Написать функцию, которая принимает путь к директории и возвращает список с наименованиями файлов, которые хранятся в полученной директории.

12. Написать функцию, которая принимает путь к директории и наименование файла и возвращает полный путь к полученному файлу.

13. Написать функцию, которая принимает список путей и возвращает число путей, которые являются директорией.

14. Написать функцию, которая принимает путь к директории и удаляет в ней все файлы, а вместо них создает директории с тем же наименованием, что и удаленные файлы без значения расширения.

15. Написать функцию относительно задания 8, которая принимает созданные 20 объектов *datetime* и создает для значений года, месяца и дня вложенные директории. Например, значение 2019-09-02 создаст три директории. Корневой будет 2019, внутри неё 09 и т.д. Стоит отметить, что значение другого объекта 2019-07-02 создаст в ранее созданной директории 2019 директорию с наименованием 07.

16. Написать функцию, которая принимает список списков и создает структуру вложенных директорий и текстовых файлов относительно входящего списка, где наименованием директории будет являться номер вложенного списка, а наименованием файла значение элемента списка. Например, для списка вида [1,[1,2]] должны быть созданы три директории и три файла, где корневая директория будет хранить файл 1.txt и директорию под именем 1, в которой будут находиться файлы 1.txt и 2.txt.

РАБОТА С ФАЙЛАМИ. РАЗРАБОТКА СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА. ВЫВОД ФОРМАТИРОВАННЫХ ДАННЫХ В ФОРМАТЕ JSON

Цель работы: изучить работу с файлами с помощью функций из стандартной библиотеки; рассмотреть понятие синтаксического анализа текста и ознакомиться с его применением на языке *Python*; изучить возможности взаимодействия *Python* с форматом хранения данных *JSON*.

Краткая теория

Работа с файлами

Python поддерживает работу с множеством различных типов файлов. Условно их можно разделить на текстовые и бинарные. Текстовые файлы – это те файлы, которые хранят данные в текстовом виде, а бинарные – это, к примеру видео, изображение и прочее.

Python позволяет работать с файлами двумя способами:

- непосредственного открытия и закрытия файла с помощью встроенных функций ***open(path, mode)*** и ***close()***;
- с применением конструкции ***with***.

Чтение файла и запись в файл осуществляются с помощью методов ***read()*** и ***write()*** соответственно. *Python* позволяет работать с файлом в определенных режимах, передаваемых функции ***open*** в качестве строкового аргумента для параметра ***mode***:

1) «***r***» (*Read*) – файл открывается для чтения. Если файл не найден, то генерируется исключение *FileNotFoundError*;

2) «***w***» (*Write*) – файл открывается для записи. В любом случае создается новый. Следовательно, старые данные будут удалены;

3) «***a***» (*Append*) – файл открывается для дополнительной записи. Если файл отсутствует, то создается новый. При существовании такого файла, данные запишутся в конец;

4) «*b*» (*Binary*) – используется для работы с бинарными файлами и применяется вместе с другими режимами («*wb*», «*rb*», «*ab*»);

5) «*+*» – файл открывается одновременно на чтение и запись. Применяется вместе с другими режимами («*r+*», «*a+*», «*wb+*» и т.д.).

Стоит выделить комбинации «*r+*» и «*w+*» при таких режимах, оба файла будут открыты на одновременное чтение и запись, но в первом случае при отсутствии файла возникнет исключение, а во втором создастся новый файл.

Как было отмечено ранее, для работы с файлами можно использовать встроенные функции *open()* и *close()* и конструкцию *with*. В первом случае стоит учитывать, что во время работы с файлами могут быть сгенерированы различные исключения, что может привести к не выполнению функции *close()* и соответственно не закрытию файла. Для предотвращения данной ситуации применяется конструкция *try*, где в блоке *finally* обязательно вызовется функция *close()*. Однако, чтобы избежать данных дополнительных действий существует конструкция *with*, которая внутри себя уже реализует обработку исключений и обязательного закрытия файла. На рис. 5.1 приведена стандартная работа с файлом и аналогичная работа с файлом с использованием конструкции *with*.

```
base_path = r'C:\Users\sulyz\Documents\python-tutorial\Laboratory5\test-text-file.txt'

# Стандартная работа с файлом с обработкой исключений
file = open(base_path, 'w', encoding="utf-8")

try:
    file.write("Программирование на Python!")
except Exception as exp:
    print(exp)
finally:
    file.close()

# Аналогичная работа с файлом, используя with
with open(base_path, 'w', encoding="utf-8") as test_file:
    test_file.write("Программирование на Python!")
```

Рис. 5.1. Пример работы с файлом с обработкой исключений и использованием конструкции *with*

Из рис. 5.1 видно, что при вызове функции *open()* указывается третий аргумент для параметра *encoding* (кодировки файла). Это необходимо, так как в файл записывается кириллица.

Синтаксический анализ

Часто приходится из текстового документа выделять конкретную информацию для её организации, упорядочивания и представления в удобной и читаемой форме, но при этом, не работая с ресурсом вручную. Как правило, это большие текстовые данные, которые тяжело и долго обрабатывать вручную, однако это может быть и обработка обновляемой информации (например, расписания занятий с сайта). Для этого применяют синтаксический анализ.

Синтаксический анализ (СА) – это процесс анализа информации и сопоставление последовательности токенов (лексем) естественного или формального языка с его формальной грамматикой. Как правило, СА работает вместе с лексическим анализом.

Лексический анализ (ЛА) – это процесс аналитического разбора входных символов и разбивка их на структурные единицы языка, называемые токенами или лексемами [12].

В данном пособии не рассматривается углубленное понимание СА и ЛА, а дается только пример написания собственного синтаксического анализатора. Для примера разработаем синтаксический анализатор веб-страницы стоимости криптовалют (<https://coinmarketcap.com/>), исходный код которого приведен на рис. 5.2. На рис. 5.3 приведен фрагмент *HTML*-документа с веб-страницы.

```
from bs4 import BeautifulSoup

with open('crypto-html.html', 'r', encoding='UTF-8') as schedule_file:
    html_content = schedule_file.read()

soup = BeautifulSoup(html_content)
lines = soup.find('table', id='currencies').find('tbody').find_all('tr')

cryptocurrency_data = []
for line in lines:
    currency_name = line.find('a', {'class': 'currency-name-container'}).getText()
    market_cap = line.find('td', {'class': 'market-cap'}).getText().replace('\n', '')
    price = line.find('a', {'class': 'price'}).getText()
    change = line.find('td', {'class': 'percent-change'}).getText()

    cryptocurrency_data.append({'currency_name': currency_name, 'market_cap': market_cap,
                                'price': price, 'change(24h)': change})

print(cryptocurrency_data)
```

Рис. 5.2. Исходный код синтаксического анализатора

Как показано на рис. 5.2, на выходе программы выводится список словарей с данными по стоимости криптовалют. Данная

программа работает с *HTML* документом с помощью стороннего модуля *bs*, который можно установить через окно консоли с помощью команды *pip install beautifulsoup4*.

```
<tr id="id-ripple" class="">
  <td class="text-center">
    3
  </td>
  <td class="no-wrap currency-name" data-sort="XRP">
    
    <span class="currency-symbol visible-xs"><a class="link-secondary" href="/currencies/ripple/">XRP</a></span>
    <br class="visible-xs">
    <a class="currency-name-container link-secondary" href="/currencies/ripple/">XRP</a>
  </td>
  <td class="no-wrap market-cap text-right" data-usd="12847482054.6" data-btc="1383539.58923" data-sort="12847482054.6">
    $12,847,482,055
  </td>
  <td class="no-wrap text-right" data-sort="0.297064715649">
    <a href="/currencies/ripple/#markets" class="price" data-usd="0.297064715649" data-btc="3.19907661998e-05">$0.297065</a>
  </td>
  <td class="no-wrap text-right" data-sort="2153334120.77">
    <a href="/currencies/ripple/#markets" class="volume" data-usd="2153334120.77" data-btc="231891.587182">$2,153,334,121</a>
  </td>
  <td class="no-wrap text-right circulating-supply" data-sort="43248091671.0">
    <span data-supply="43248091671.0">
      <span data-supply-container>43,248,091,671</span>
    </span>
  </td>
</tr>
```

Рис. 5.3. Фрагмент *HTML* кода страницы стоимости криптовалют

Программа «достаёт» текст, хранящийся в определенных *HTML*-тегах. Поиск необходимых тегов осуществляется с помощью методов *find()* и *find_all()*, которые принимают наименование тега, идентификатор (необязательно) и наименование классов в виде словаря, которые установлены на искомом теге.

Таким образом, с помощью использования синтаксического анализа можно получать необходимую информацию как с различных веб-ресурсов, так и из обычных текстовых файлов.

Формат *JSON*

Json (*JavaScript Object Notation*) – это простой формат обмена данными. Он прост для чтения и записи как человеком, так и программными средствами. Файлы, содержащие *Json*, имеют расширение *.json*. *Json* основан на двух структурах данных:

- коллекция пар ключ-значение. В *Python* такой структурой называется словарь;
- упорядоченный список значений. В *Python* – список.

На сегодняшний день *Json* является самым популярным форматом передачи данных. Стандарт принят и поддерживается всеми крупными компаниями. Однако также используются и другие форматы передачи данных, такие как *XML*, *YAML* и другие. Но они сейчас не так популярны. В настоящий момент по-

что все приложения, которые передают данные через Интернет, делают это с помощью *Json*. Пример содержания *Json* файла приведен на рис. 5.4.

```
1 {  
2   "name_lab_manual": "ПРОГРАММИРОВАНИЕ НА  
3                       ЯЗЫКЕ PYTHONПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ PYTHON",  
4  
5   "authors": ["Широбокова С.Н.", "Кацупеев А.А.", "Сулыз А.В."]   
6 }
```

Рис. 5.4. Пример содержания *Json* файла

Как видно из рис. 5.4, для хранения имени методического пособия указывается значение и ключ, как это используется в словарях. Для хранения авторов используется список, который выступает в виде значения для ключа.

Ранее было указано, что модуль для работы с данным форматом данных входит в стандартную библиотеку *Python*. Рассмотрим данный модуль подробнее.

Модуль *Json*

- ***dump(obj, file)*** – метод предназначен для процесса сериализации (преобразование) объекта *obj* в формат *Json* и записи его в файл *file*;
- ***dumps(obj)*** – метод предназначен для процесса сериализации (преобразование) объекта *obj* в формат *Json* в виде строки;
- ***load(file)*** – метод предназначен для процесса десериализации (обратного преобразования) файла *file*, содержащего *Json* в объект *Python*;
- ***loads(str)*** – метод предназначен для процесса десериализации *Json* строки *str* в объект *Python*.

Стоит отметить, что методы, описанные выше, содержат большее количество параметров. С ними можно ознакомиться в официальной документации *Python 3*. На основе разработанного ранее синтаксического анализатора, рассмотрим сериализацию и десериализацию данных. Исходный код сериализации и десериализации представлен на рис. 5.5 (дополнение к исходному коду рис. 5.2).

Таким образом, с помощью использования СА и *Json* можно обрабатывать различные веб-ресурсы и данные, а затем в удобной форме передавать их через Интернет. Такой работы

придерживаются почти все современные приложения с многоуровневой архитектурой. Например, при разработке мобильных приложений, когда СА работает на стороне сервера, а мобильное приложение в ответ на запрос о получении данных получает их в формате *Json*, который поддерживается почти всеми современными языками программирования.

```
print(cryptocurrency_data)

import json

serialization_data = json.dumps(cryptocurrency_data)
print(serialization_data)

deserialization_data = json.loads(serialization_data)
print(deserialization_data)
```

Рис. 5.5. Исходный код сериализации и десериализации с помощью модуля *Json*

Варианты заданий

1. Написать функцию, которая принимает *Json* строку и конвертирует её в словарь.
2. Написать функцию, которая принимает путь к файлу и количество строк, которые требуется прочитать и возвращает считанные строки в файле.
3. Написать программу, которая принимает путь к файлу и возвращает наиболее длинное слово из него.
4. Написать функцию, которая принимает путь к файлу, текст и номер строки и записывает в файл полученный текст в указанный номер строки.
5. Написать программу, которая принимает путь к файлу и возвращает *True* или *False* в зависимости от того, доступен ли файл для чтения и записи или нет.
6. Написать функцию, которая принимает путь к *HTML* файлу и *html* тег («*p*», «*h1*», «*article*» и др.) и возвращает количество повторений полученного тега в файле с учетом того, что требуется вернуть только количество тегов, который имеет открывающую и закрывающую часть.
7. Написать функцию, которая принимает путь к директории, путь к файлу и записывает в файл информацию обо всех файлах в директории (формат файла, размер, дата создания).

Информация о файлах должна быть пронумерована, а каждый параметр должен быть помечен. Например, «1. *Test*; Расширение файла: *py*; Дата создания: 2019-12-12...» и т.д.

8. Написать функцию, которая принимает путь к двум файлам и возвращает количество одинаковых предложений в файлах.

9. Написать функцию, которая принимает путь к файлу и параметр *x*, который может являться строкой или списком, и возвращает частоту повторений параметра *x* в строке. В случае, когда параметром *x* является список, следует вернуть словарь, в котором в качестве ключей будут искомые строки, а их значениями частота повторений.

10. Написать функцию, которая принимает путь к *HTML* и путь к *CSS* файлам и возвращает словарь, в котором ключами выступают теги, идентификаторы или классы в файле *CSS*, а значениями список списков, где первым элементом внутреннего списка будет наименование тега, которые попадают под стили, указанные в файле *CSS*, а вторым – номер строк, в которых они находятся. Например, `{'#inline-text': [['h1', 29], ['p', 50]]}`.

11. Написать функцию, которая принимает путь к файлу и возвращает словарь, в котором в качестве значений выступают строки, а их ключами – количество символов русского алфавита. Символы русского следует генерировать в процессе выполнения функции, используя *build-in* функции *ord()* и *chr()*.

12. Написать функцию, принимающую словарь, который может содержать любые элементы и конвертирует в *Json* только те элементы, значения которых являются списками, содержащими только целочисленные значения.

13. Написать функцию, которая принимает ссылку на страницу с расписанием студентов ЮРГПУ(НПИ) и возвращает путь к файлу *Json*, в котором все возможные данные о расписании по полученной ссылке, в том числе курсе и коде учебной группы. Можно использовать специализированный синтаксический анализатор для *HTML* (*BeautifulSoup*, *lxml* и другие).

Лабораторная работа №6

РАЗРАБОТКА ПРИЛОЖЕНИЯ РАБОТЫ С БАЗОЙ ДАННЫХ

Цель работы: изучить возможности взаимодействия *Python* с реляционными базами данных с помощью *DB-API 2.0*.

Краткая теория

Почти каждая программа, которая работает с данными, требует места для их хранения. В качестве такого хранилища могут выступать обычные текстовые файлы, файлы *JSON* или *XML*. Однако для хранения большого количества данных и удобной организации работы с ними существуют СУБД. В данной лабораторной работе не рассматриваются понятия и термины баз данных, синтаксис *SQL*. Здесь будут рассмотрены механизмы для работы с базами данных с помощью *Python*.

DB-API 2.0

Несмотря на существующий стандарт *SQL (ISO/IEC 9075)* каждая СУБД имеет ряд отличий от других. Для того, чтобы программисту не вникать в их реализации в *Python* существует общее *API*, описанное стандартом *PEP 249*. Стоит учесть, что *PEP 249* это всего лишь спецификация, реализация которой выполняется самостоятельно. Однако существуют уже готовые реализации данного *API* для ряда СУБД. Стоит отметить, что *Python* имеет в качестве встроенного модуля одну из реализаций данного *API* для работы с *SQLite*. Ниже приведены некоторые реализации для других СУБД:

- ***Oracle***: *cx_Oracle*, *mxODBC*, *pyodbc*;
- ***MySQL***: *mysql-python*, *PyMySQL*;
- ***MS SQL Server***: *adodbapi*, *pymssql*, *mxODBC*, *pyodbc*;
- ***PostgreSQL***: *psycopg2*, *txpostgres*.

Для примера работы с *API* рассмотрим работу с данными небольшой базы данных библиотеки. На рис. 6.1 приведена *ERD* диаграмма рассматриваемой базы данных.

На рис. 6.2 показан пример выполнения запроса к таблице *Author* с помощью реализации рассматриваемого *API*, а на рис. 6.3 приведен вывод результата запроса.

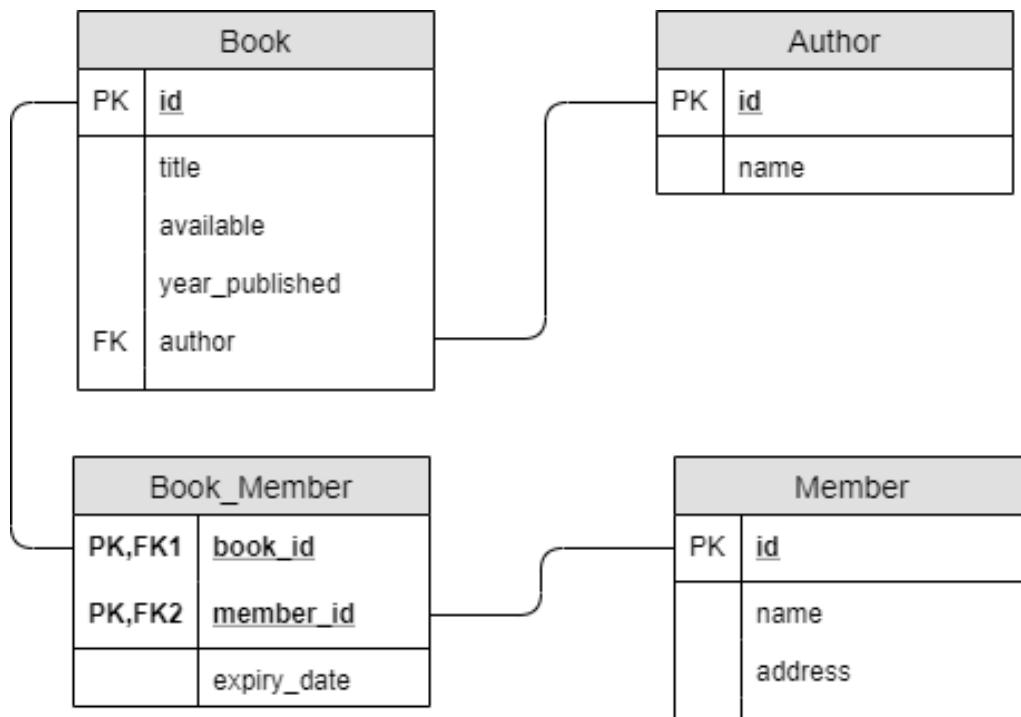


Рис. 6.1. ERD диаграмма рассматриваемой базы данных

```

import sqlite3

conn = sqlite3.connect('librarydb-sqlite.db')
cursor = conn.cursor()

cursor.execute("SELECT * FROM Author")
results = cursor.fetchall()

print(results)

conn.close()
  
```

Рис. 6.2. Пример выполнения запроса к таблице *Author*

```

(1, 'Лев Николаевич Толстой')
(2, 'Александр Сергеевич Пушкин')
(3, 'Михаил Юрьевич Лермонтов')
>>>
  
```

Рис. 6.3. Вывод результата запроса к таблице *Author*

Для подключения реализации рассматриваемого API при работе с *Sqlite* необходимо импортировать пакет *sqlite3* из стандартной библиотеки *Python*. Чтобы открыть соединение с базой необходимо использовать функцию *connect(db_file_path)*, которая в качестве значения аргумента принимает путь к файлу

SQLite. При работе с другими СУБД потребуется указать дополнительные параметры функции, такие как хост, имя пользователя, пароль и наименование базы.

Для выполнения запросов и получения результатов требуется использовать специальный объект *Cursor*, который можно получить, вызвав метод *cursor()* из объекта *Connection*, который возвращается из функции *connect()*.

Вызов метода *execute(query)* для объекта *Cursor* выполняет запрос и в случае наличия результата возвращает ответ, который можно получить методом *fetchall()*. Стоит отметить, что данный метод возвращает значение только один раз, повторное обращение к нему без предварительного запроса вернет пустой список.

Для закрытия соединения с базой данных используется метод *close()* объекта *Connection*.

На рис. 6.4 приведен пример запроса на несколько строк, осуществляемый с помощью тройных кавычек.

```
import sqlite3

conn = sqlite3.connect('librarydb-sqlite.db')
cursor = conn.cursor()

cursor.execute("""
    SELECT *
    FROM BOOK
    ORDER BY title
""")

results = cursor.fetchall()

print(results)

conn.close()
```

Рис. 6.4. Пример запроса на несколько строк

На рис. 6.5 представлен пример записи данных в базу данных, а на рис. 6.6 пример использования подстановки именованного значения в запрос.

Как видно из рис. 6.5, запись в базу данных с помощью *API* осуществляется аналогично предыдущему примеру за исключением того, что после запроса на запись требуется вызвать метод *commit()* объекта *Connection* для сохранения транзакции.

```
import sqlite3

conn = sqlite3.connect('librarydb-sqlite.db')
cursor = conn.cursor()

cursor.execute("""
    INSERT INTO member('name', 'address')
    VALUES('ИВАНОВ ИВАН ИВАНОВИЧ', 'Ростов-на-Дону, пр-т. Коммунистический, 16')
""")

conn.commit()
|
conn.close()
```

Рис. 6.5. Пример записи данных в базу данных

```
import sqlite3
import datetime

conn = sqlite3.connect('librarydb-sqlite.db')
cursor = conn.cursor()

cursor.execute("""
    INSERT INTO book_member
    VALUES(1, 1, :expiry_date)""",
           {'expiry_date': datetime.date(2019, 11, 12)} )

conn.commit()

conn.close()
```

Рис. 6.6. Пример записи данных в базу данных с именованной подстановкой значения

На рис. 6.6 показано использование именованных замен при запросе. Для этого требуется использовать двоеточие перед наименованием метки, и передать словарь со значением метки вторым аргументом для метода *execute()*.

Варианты заданий

1. Написать функцию, которая принимает наименование таблицы, поля и его значение и возвращает идентификатор записи, в которой значение полученного поля соответствует переданному функции, или возвращает *None*.

2. Написать функцию, которая принимает наименование таблицы и выводит количество одинаковых значений в её полях и наиболее часто повторяющееся значение полей.

3. Написать функцию, которая возвращает наименование всех таблиц связи, которые находятся в базе данных, если они существуют. Стоит отметить, что в данном задании предполагается, что таблицы связи именуются склеиванием имени двух таблиц через знак нижнего подчеркивания «_».

4. Написать функцию, которая принимает наименование таблицы и список списков, которые содержат данные, и обновляет данные полученной таблицы на указанные во внутренних списках, где в качестве идентификаторов записей следует принимать номер внутреннего списка плюс единица, т.к. идентификаторы записей СУБД нумеруются с единицы.

5. Написать функцию, которая принимает словарь, в котором ключами являются наименования таблиц, а значениями список списков, содержащих данные для таблицы, и выполняет вставку полученных данных в указанные таблицы.

6. Написать функцию, которая принимает наименование таблицы, имя поля и возвращает все записи по полученному полю из указанной таблицы.

7. Написать функцию, которая переводит структуру всех таблиц существующей базы данных, в формат *Json*.

8. Написать функцию, которая принимает путь к файлу *Json*, который содержит описание таблиц и их полей, и создает новые таблицы, указанные в полученном файле.

9. Написать функцию, которая принимает путь к файлу с расширением *.txt*, который содержит данные для заполнения таблицы, и по указанным в файле данным создает таблицу или таблицы и заполняет их данными.

10. Написать функцию, которая строит *ER* модель базы данных. Можно сгенерировать схему в формате *HTML* или *XML*, либо воспользоваться сторонними приложениями и библиотеками.

Лабораторная работа №7

РАЗРАБОТКА ПРИЛОЖЕНИЯ С ИСПОЛЬЗОВАНИЕМ ООП

Цель работы: ознакомиться с методологией объектно-ориентированного программирования, изучить реализацию данной методологии в языке *Python 3*.

Краткая теория

Объектно-ориентированное программирование (ООП) – это одна из методологий (парадигм) программирования (структурное, функциональное), где компоненты программы представляются в виде объектов. Объект – это экземпляр класса, имеющего поля и методы.

Большинство современных языков программирования, таких как *Java*, *C++*, *C#* и другие, реализуют данный подход, в том числе и *Python*. В данной лабораторной работе не рассматривается теория ООП, а только реализация данной методологии в *Python 3*.

Реализация ООП в *Python 3*

Для создания класса требуется указать ключевое слово *class* указать имя создаваемого класса и поставить знак двоеточие «:». Для указания полей и методов, нужно в теле класса создать переменные и методы соответственно. На рис. 7.1 приведен пример простого класса.

```
class Animal:
    nameAnimal = ""
    ageAnimal = ""

    def __init__(self, nameAnimalOut):
        self.nameAnimal = nameAnimalOut

    def eat(self):
        print(self.nameAnimal + " eating...")

    def sleep(self):
        print(self.ageAnimal + " sleeping..")
```

Рис. 7.1. Пример реализации простого класса на *Python 3*

Как видно из рис. 7.1, в качестве первого аргумента для всех методов, приведенных в классе *Animal*, является *self*, который хранит ссылку на объект класса.

Конструктор класса – это метод, который вызывается в начале создания объекта. В *Python 3* для этого существует приведенный на рис. 7.1 метод `__init__()`. Создание объекта класса происходит путем указания имени класса с круглыми скобками «()». Следует отметить, что для создания полей объекта не обязательно объявлять их в теле класса, достаточно объявить их в конструкторе, как показано на рис. 7.2.

```
class Animal:

    def __init__(self, nameAnimalOut):
        self.nameAnimal = nameAnimalOut
        self.ageAnimal = 1

    def eat(self):
        print(self.nameAnimal + " eating...")

    def sleep(self):
        print(str(self.ageAnimal) + " sleeping..")
```

Рис. 7.2. Пример реализации простого класса на *Python 3* с инициализацией переменных в конструкторе

Модификаторы доступа. Инкапсуляция

В *Python* существуют три вида модификаторов доступа: *public*, *protected*, *private*. Модификатор доступа *public* разрешает доступ к переменным из любой точки вне и внутри класса. Модификатор доступа *private* допускает обращения к переменным только внутри класса. Модификатор доступа *protected* разрешает доступ из любой точки вне и внутри класса, внутри пакета, а также в классах-наследниках.

Инкапсуляция – это сокрытие данных одного класса от прямого обращения к его данным другими классами. Суть заключается в том, чтобы предоставлять доступ к переменным класса через методы геттеры и сеттеры, в которых может происходить логика, связанная с работой защищаемой переменной.

Для реализации модификатора доступа *protected* требуется перед наименованием переменной поставить один знак нижнего подчеркивания *self._nameAnimal*, а для модификатора доступа *private* – два знака нижнего подчеркивания *self.__ageAnimal*. На рис. 7.3 представлен пример использования инкапсуляции в *Python 3*.

```
class Factory:

    def __init__(self):
        self.__workers = []

    def get_workers(self):
        return [w for w in self.__workers]

    def add_workers(self, worker):
        self.__workers.append(worker)

factory_obj = Factory()
print(factory_obj.get_workers())

factory_obj.add_workers('Иван Петрович')
print(factory_obj.get_workers())

factory_obj.__workers
```

Рис. 7.3. Пример применения инкапсуляции в *Python*

Из рис. 7.3 видно, что для объявления переменной *workers* используется конструктор, а сама переменная объявляется с модификатором доступа *private*. Далее создается метод для получения значений из данной переменной, однако, так как эта переменная хранит в себе список, с помощью метода *get_workers* передается копия данного списка. Таким образом, классы-пользователи данной переменной не смогут её изменить. Такое состояние называется иммутабельностью. В данном случае при прямом обращении из точки кода вне класса *Factory* к переменной *workers* возникнет исключение *AttributeError*. Однако данное исключение можно обойти и получить доступ к переменной с модификатором доступа *private* с помощью рефлексии. Подробнее с *Reflection API* в *Python 3* можно ознакомиться по ссылке [13].

Вывод вышеприведенного кода представлен на рис. 7.4.

```
[  
['Иван Петрович']  
Traceback (most recent call last):  
  File "C:/Users/sulyz/Documents/python-tutorial/Lab7-source/incupsulation.py",  
  line 19, in <module>  
    factory_obj.__workers  
AttributeError: 'Factory' object has no attribute '__workers'
```

Рис. 7.4. Результат работы кода, представленного на рисунке 7.3

Наследование

Наследование – это предоставление характеристик класса-родителя классу-потомку. Данный механизм позволяет в несколько раз сократить количество кода за счет его повторного использования. Стоит также отметить, что в *Python* существует поддержка множественного наследования.

Для реализации наследования в *Python 3* необходимо при объявлении класса в круглых скобках указать классы-родителей. На рис. 7.5 приведен пример использования наследования в *Python 3*.

```
class Animal:  
  
    def __init__(self, name):  
        self.name_animal = name  
  
    def eat(self):  
        print(self.name_animal + ' eating...')  
  
    def sleep(self):  
        print(self.name_animal + ' sleeping...')  
  
class Cat(Animal):  
  
    def meow(self):  
        print(self.name_animal + ' say meow')  
  
class Dog(Animal):  
  
    def woof(self):  
        print(self.name_animal + ' say woof')
```

Рис. 7.5. Пример использования наследования в *Python 3*

Как видно из рис. 7.5, классы *Cat* и *Dog* являются классами-наследниками класса *Animal*, что позволяет этим классам использовать все методы и поля класса-родителя, включая кон-

структор. На рис. 7.6 приведен пример использования классов-наследников, а на рис. 7.7 результат вызова методов родителя из классов-наследников.

```
cat = Cat('Piter')
cat.eat()
cat.meow()

dog = Dog('Tommy')
dog.sleep()
dog.woof()
```

Рис. 7.6. Пример использования классов-наследников

```
Piter eating...
Piter say meow
Tommy sleeping...
Tommy say woof
```

Рис. 7.7. Результат вызова методов из классов-наследников

Полиморфизм

Полиморфизм заключается в способности объекта вести себя по-разному. Под полиморфизмом рассмотрим перегрузку и переопределение методов.

Перегрузка метода – это способность метода вести себя по-разному при определенном наборе параметров. Переопределение метода заключается в изменении логики метода с таким же наименованием и количеством параметров, что и метод в классе-родителе.

На рис. 7.8 приведен пример перегруженного и переопределенного методов. В случае, представленном на рис. 7.8, перегрузка конструктора позволяет инициализировать класс *Cat* несколькими способами (с указанием породы и имени кошки или только имени). Переопределение же метода *eat()* позволяет заменить реализацию метода класса-родителя реализацией класса-наследника.

Таким образом, использование ООП не только позволяет создавать более гибкий и читаемый код, но и вносит дополнительный уровень безопасности. Однако стоит учитывать, что у

данного подхода есть и недостаток, заключающийся в увеличении сложности программы.

```
class Animal:

    def __init__(self, name):
        self.name_animal = name

    def eat(self):
        print(self.name_animal + ' eating...')

    def sleep(self):
        print(self.name_animal + ' sleeping...')

class Cat(Animal):

    def __init__(self, name, breed_cat): # перегрузка
        self.name_animal = name
        self.breed_cat = breed_cat

    def eat(self): # переопределение
        print('Cat ' + self.name_animal + ' purring and eating...')

    def meow(self):
        print(self.name_animal + ' say meow')
```

Рис. 7.8. Пример перегрузки и переопределение методов в *Python 3*

Варианты заданий

1. Написать родительский и дочерний классы с методами «*display*». В каждом методе в консоль выводится различная строка.
2. Написать класс *Exception* с возможностью передачи в него сообщения и реализовать класс, в котором будет пробрасываться данный *Exception*.
3. Написать класс, который реализует паттерн *Singleton*.
4. Написать класс, который содержит два метода «*get_string*» и «*print_upper_string*», где первый метод принимает строку, а второй выводит данную строку в верхнем регистре.
5. Написать класс *UserLanguagePreference*, который в конструкторе принимает список языков в виде строки, которые использует пользователей и содержит метод *add_lang(lang_str)*, который добавляет язык в список, если его там не существует.

Данный класс использует инкапсуляцию, для получения используемого списка.

6. Написать класс *Shape*, который является родительским для класса *Square*, который содержит конструктор, принимающий длину. Оба класса содержат метод *area()* для расчета площади. Причем класс *Shape* имеет площадь равную нулю.

7. Написать класс *Point*, который в конструкторе принимает координаты точки и содержит методы *show()*, *move()* и *dist()*, где первый метод возвращает координаты точки, второй принимает значения, на которые нужно сместить координаты точки, и последний выводит расстояние по следующей формуле:

$$d = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}.$$

8. Написать класс *ORMMapper*, который в конструкторе принимает имя другого существующего класса, который содержит только поля и с помощью метода *convert_to_db()* создает на основе структуры полученного класса таблицу в базе данных.

9. Написать класс *Publisher* и несколько различных классов с постфиксом *Subscriber*. Реализовать между данными классами отношения по паттерну *Observer*.

10. Написать класс *ChartDrawer*, который принимает список списков, содержащих координаты точек и содержит методы *add_point(x, y)* для добавления новой точки в список, *remove_point(x, y)* для удаления точек с указанными координатами. Класс должен правильно использовать инкапсуляцию.

Лабораторная работа №8

ОБРАБОТКА ИЗОБРАЖЕНИЙ С ПРИМЕНЕНИЕМ БИБЛИОТЕКИ *PIL*

Цель работы: рассмотреть возможности работы библиотеки *PIL*, изучить её базовые инструменты для обработки изображений.

Краткая теория

Для *Python* существует огромное количество полезных библиотек, одной из таких является библиотека для обработки изображений – *Python Image Library (PIL)*. Она обеспечивает работу с разнообразными форматами файлов и мощные инструменты для обработки изображений.

С помощью *PIL* можно создавать миниатюры изображений (*thumbnails*), конвертировать изображения в различные форматы, изменять размер, вращать изображение, а также с помощью данной библиотеки можно реализовать функцию отправки изображения на печать. Для того, чтобы загрузить *PIL* в командной строке или терминале, требуется ввести следующую команду: *pip install Pillow*.

Основные функции для работы с изображениями

Чтобы загрузить изображение из файла, требуется вызвать метод *open(path_to_file)* класса *Image*, который можно импортировать с помощью команды *from PIL import Image*, передав в качестве аргумента для параметра *path_to_file* путь к файлу с изображением. В случае успешного чтения файла метод *open()* вернет объект класса *Image*, в противном случае возникнет исключение *IOError*. Из полученного объекта, обращаясь к полям *format*, *size* и *mode*, можно получить формат изображения, его размер (ширину и высоту) в пикселях и схему формирования цвета (*Luminance*, *RGB*, *CMYK*) соответственно.

Для того, чтобы конвертировать изображение в другой формат, достаточно его сохранить заново с другим расширением с помощью метода *save(new_file_name)*, где в качестве аргумента для параметра *new_file_name* требуется указать путь но-

вого изображения с наименованием и требуемым расширением. Процесс открытия и конвертирования изображения представлен на рис. 8.1.

В примере на рис. 8.1 опущено значение переменной *basic_image_path*, значением которой является путь к папке с файлом изображения.

```
>>>
>>>
>>> from PIL import Image
>>> image_obj = Image.open(basic_image_path + '\WallpapersOrigin.jpg')
>>> print(image_obj.format, image_obj.size, image_obj.mode)
JPEG (4267, 2400) RGB
>>> png_image_path = basic_image_path + '\WallpapersOrigin.png'
>>> image_obj.save(png_image_path)
>>>
>>>
```

Рис. 8.1. Пример конвертирования изображения с помощью *PIL*

Другой часто встречающейся задачей является создание миниатюры изображения для быстроты их загрузки на экранную форму или страницу сайта. Для этого требуется предварительно создать кортеж, который состоит из двух элементов высоты и ширины создаваемой миниатюры, после чего для объекта класса *Image* вызвать метод *thumbnail(size)*, где в качестве аргумента параметра *size* передать созданный кортеж. Пример создания миниатюры изображения приведен на рис. 8.2 в качестве дополнения кода, приведенного на рис. 8.1.

```
>>>
>>> size = (50, 50)
>>> image_obj.thumbnail(size)
>>> image_obj.save(png_image_path)
>>>
```

Рис. 8.2. Пример создания миниатюры изображения с помощью *PIL*

Обработка изображения

Часто при обработке изображений требуется обработать его цвета. *PIL* предоставляет удобные инструменты для работы с различными фильтрами, обработки цветовых каналов изображения, изменения контраста, яркости, цветового баланса и др.

PIL предоставляет широкий спектр фильтров для обработки изображений, которыми можно воспользоваться с помощью класса *ImageFilter*. Рассмотрим обработку изображения с помощью фильтра размытие по Гауссу. Для получения объекта рассматриваемого фильтра требуется вызывать метод *GaussianBlur(radius=2)*, значением параметра которого можно изменить на желаемое число. Данный объект требуется передать в качестве аргумента методу *filter(img_filter)* объекта класса *Image*. Для того, чтобы отобразить результат, можно вызвать метод *show()* у объекта изображения. Пример применения фильтра приведен на рис. 8.3.

```
>>> from PIL import Image
>>> from PIL import ImageFilter
>>> im = Image.open(r'C:\Users\sulyz\Documents\python-tutorial\Lab8-source\WallpapersOrigin.jpg')
>>> im_filter = im.filter(ImageFilter.GaussianBlur(radius=10))
>>> im_filter.show()
>>>
>>>
```

Рис. 8.3. Пример применения фильтра размытия по Гауссу на изображение с помощью *PIL*

Для изменения контраста, яркости и цветового баланса изображения требуется использовать класс *ImageEnhance*. Пример изменения контраста и яркости изображения приведен на рис. 8.4.

```
>>> from PIL import ImageEnhance
>>> contrast = ImageEnhance.Contrast(im)
>>> contrast.enhance(0.5).show()
>>> brightness = ImageEnhance.Brightness(im)
>>> brightness.enhance(0.9).show()
```

Рис. 8.4. Изменения контраста и яркости изображения с помощью *PIL*

Таким образом, *Pillow* – это мощнейший инструмент для обработки изображений на *Python*. В данной лабораторной работе рассмотрена только небольшая часть того, что можно делать с данной библиотекой. Более подробно ознакомиться с *PIL* можно на странице с документацией данного проекта [14].

Варианты заданий

1. Написать функцию, которая принимает путь к директории и создает новую директорию по тому же пути с именем *thumbnail*, в которую записывает иконки изображений, находящихся в принятой функцией директории.
2. Написать функцию, которая принимает путь к изображению и конвертирует его в формат *.ICO*, сохраняя её по тому же пути, что и исходное изображение.
3. Написать функцию, которая принимает путь к изображению и выполняет над ней *autocontrast*, сохраняя новое изображение по тому же пути.
4. Написать функцию, которая принимает путь к изображению и создает отраженное изображение, сохраняя его по тому же пути.
5. Написать функцию, которая принимает текст и стиль шрифта в виде строки и создает изображение с полученным текстом и шрифтом.
6. Написать функцию, которая принимает путь к изображению и поворачивает его на 90 градусов, сохраняя его по тому же пути.
7. Написать функцию, которая принимает путь к директории с изображениями и список расширений (*.png*, *.jpg*, *.ico* и т.д.) и создает директории по полученному пути с наименованием расширения, в которые записывает конвертированные исходные изображения по полученным форматам.
8. Написать функцию, которая принимает путь к *tar* архиву и путь для сохранения и распаковывает из него все изображения, сохраняя их по указанному пути.
9. Написать функцию, которая принимает путь к изображению и список пикселей и заменяет полученные пиксели прозрачным цветом, сохраняя изображение в той же директории.

10. Написать функцию, которая принимает путь к изображению, искомый цвет пикселя и цвет, на который требуется его заменить, и заменяет полученный цвет на картинке цветом замены, сохраняя новое изображение в той же директории.

11. Написать класс, который содержит функцию, принимающую путь к *gif* изображению и делает его раскадровку, сохраняя изображения по тому же пути, что и *gif*. Вторая функция будет принимать директорию с изображениями и записывать их в *gif*.

12. Написать класс, который содержит функцию, выполняющую скриншот экрана и функцию, которая сохраняет изображение, находящееся в буфере, если такое существует.

13. Написать функцию, которая принимает путь к файлу с изображением и два списка координат пикселей и выполняет перемещение пикселей двух списков.

14. Написать функцию, которая принимает путь к изображениям и склеивает их в одно изображение.

Лабораторная работа №9
**РАЗРАБОТКА GUI ПРИЛОЖЕНИЯ С ПОМОЩЬЮ
ГРАФИЧЕСКИХ БИБЛИОТЕК**

Цель работы: рассмотреть возможности библиотеки *Tkinter*, её базовые виджеты и изучить основные принципы создания приложений с графическим интерфейсом с помощью данной библиотеки.

Краткая теория

Python не является популярным языком для написания приложений с графическим интерфейсом (*GUI*) под *Windows*, однако его часто используют при написании таких приложений под *Linux*. В настоящее время существует достаточно большое количество графических библиотек для *Python 3*. Например, некоторые из них:

- *Tkinter*;
- *wxPython*;
- *PyQt*;
- *PyGTK*;
- *Kivy*;
- *Flexx*.

В данной лабораторной работе не рассматривается применение всех вышеуказанных библиотек, а только работа с *Tkinter*. *Tkinter* (*Tk interface*) – это библиотека, которая поставляется вместе с интерпретатором *Python* и как и другие предназначена для создания кросс-платформенных графических интерфейсов. Стоит отметить, что она поставляется в том случае, если вместе с интерпретатором устанавливается *IDLE*, потому что, как не трудно догадаться, он разработан с помощью данной библиотеки. Для подключения библиотеки в файл скрипта достаточно выполнить команду *import tkinter*.

Tkinter предоставляет достаточное количество различных графических элементов, таких как кнопки, метки, текстовые контейнеры и другие. В рамках рассматриваемой библиотеки они называются виджетами (*widgets*). Ниже приведены некоторые из них:

- *Button*. Виджет, используемый для отображения кнопок в приложении.
- *Canvas*. Предназначен для фигур, таких как линии, овалы, треугольники и другие.
- *Checkbutton*. Реализует функции выбора вариантов с поддержкой множественного выбора. Аналогичен работе компонента *Checkbox* в других средах.
- *Frame*. Используется в качестве контейнера для других виджетов.
- *Label*. Представляет собой однострочную строковую метку. Также может содержать изображения.
- *Listbox*. Используется для отображения списков.
- *Text*. Предназначен для отображения многострочного текста.
- *tkMessageBox*. Предназначен для отображения сообщений пользователю в новом окне.
- *Radiobutton*. Аналогичен назначению виджета *Checkbutton* с учетом того, что не поддерживается множественный выбор.

Для создания простого окна требуется всего лишь создать новый объект *Tk* и для нового объекта вызывать метод *mainloop()*. Пример создания пустого окна с инициализацией виджета *Frame* приведен на рис. 9.1.

```
import tkinter

class WindowFrame(tkinter.Frame):

    def __init__(self, root):
        tkinter.Frame.__init__(self, root)
        self.root = root

root = tkinter.Tk()
app = WindowFrame(root)
root.mainloop()
```

Рис. 9.1. Пример создания пустого окна с виджетом *Frame* с помощью *Tkinter*

Как видно из рис. 9.1, виджеты можно объявить в конструкторе нового класса, обязательно передав ему объект окна (*Tk*). Стоит отметить, что созданный класс *WindowFrame* явля-

ется наследником класса виджета *Frame*, поэтому при инициализации создаваемого класса требуется инициализировать и родительский класс. На рис. 9.2 приведен пример создания кнопки на разработанном ранее окне.

Как видно из рис. 9.2, были добавлены два метода *init_frame()* и *init_quit_button()* для настройки виджета *Frame* и создания и настройки виджета кнопки соответственно. Метод *pack()*, который имеется у всех виджет-объектов, представляет собой один из трех менеджеров геометрии: упаковщик, сетка и размещение по координатам.

```
class WindowFrame(tkinter.Frame):

    def __init__(self, root):
        tkinter.Frame.__init__(self, root)
        self.root = root

        self.init_frame()
        self.init_quit_button()

    def init_frame(self):
        self.pack(fill=tkinter.BOTH, expand=1)

    def init_quit_button(self):
        btn = tkinter.Button(self, text="Выход")
        btn.place(x=0, y=0)
```

Рис. 9.2. Пример создания кнопки на окне с помощью *Tkinter*

У метода *pack()* есть параметр *side* (сторона), который принимает одно из четырех значений-констант *tkinter* — *TOP*, *BOTTOM*, *LEFT*, *RIGHT* (верх, низ, лево, право). По умолчанию, когда в *pack()* не указывается *side*, его значение равняется *TOP*. Из-за этого виджеты располагаются вертикально. Параметр *fill* заставляет виджет заполнить всё свободное пространство как по одной оси, так и по обеим (*X*, *Y*, *BOTH*).

Для реализации виджета кнопки, как показано на рис. 9.2, требуется создать объект данного виджета, передав ему в качестве первого аргумента объект контейнера, к которому он будет привязан. В приведенном случае таким контейнером является сам объект класса *WindowFrame*, т.к. он является наследником виджета *Frame*, о котором было написано ранее.

Можно заметить, что кнопка выхода, реализованная на рис. 9.2, на самом деле не работает. Это связано с тем, что к ней не привязан никакой слушатель, который бы обрабатывал нажатия. Для привязки слушателя к виджету кнопки требуется при создании объекта этого виджета передать в качестве аргумента для параметра *command* наименование метода-слушателя, как показано на рис. 9.3.

```
def init_quit_button(self):
    btn = tkinter.Button(self,
                          text="Выход",
                          command=self.exit_command)

    btn.place(x=0, y=0)

def exit_command(self):
    exit()
```

Рис. 9.3. Пример установки слушателя на кнопку с помощью *Tkinter*

После установки слушателя на рис. 9.3 кнопка «Выход» начала выполнять свою функцию закрытия приложения. Однако можно было заметить, что на данном этапе при открытии окна оно запускается в свернутой форме. Для решения этой проблемы требуется вызывать метод *geometry(size)* объекта окна, передав в качестве аргумента для параметра *size* размер окна в виде строки, например «400x300».

Проблему при создании приложений с графическим интерфейсом может вызывать тот факт, что они работают в одном потоке, т.е. все операции, обрабатываемые в таком приложении, будут выполняться последовательно. Это приведет к тому, что, например, при написании программы, которая будет обращаться к веб-серверу или выполнять долгие математические расчеты графический интерфейс будет неактивен. Для этого требуется воспользоваться модулем *threading*.

Таким образом, *Tkinter* представляет собой удобный и мощный инструмент для создания кроссплатформенных приложений с графическим интерфейсом «из коробки». В данной лабораторной работе рассмотрены только базовые элементы, которые необходимы для начала работы с *Tkinter*, более же подробно можно ознакомиться с документацией по ссылке [15].

Варианты заданий

1. Написать *GUI* приложение, которое имеет текстовое поле и кнопку для сохранения текста, при нажатии на которую открывается модальное окно указания пути для сохранения текстового файла.

2. Написать *GUI* приложение, которые позволяет просматривать *PDF* файлы.

3. Написать *GUI* приложение, которое имеет кнопку «Сделать скриншот», по нажатию на которую создается скриншот экрана и вставляет в окно приложение и появляются возможности настройки яркости изображения, некоторые фильтры и поворот изображения. По нажатию на кнопку «Сохранять» должно сохраняться сконфигурированное изображение.

4. Написать *GUI* приложение, которое представляет собой упрощенный файловый менеджер, с возможностью создания, удаления и переименования директорий и файлов.

5. Написать *GUI* приложение, которое представляет собой галерею изображений с возможностью просмотра плитки изображений, каждого изображения по отдельности и добавления новых изображений в галерею.

6. Написать *GUI* приложение, которое представляет собой библиотеку с поиском книг по автору и наименованию, добавлению новых книг и чтению книг в форматах *epub*, *fb2* и *pdf*.

7. Написать *GUI* приложение, которое позволяет рисовать в определенном поле с помощью курсора мыши. В приложении также можно выбрать цвет и толщину кисти.

8. Написать *GUI* приложение, которое представляет собой несложный мессенджер, работающий с другим пользователем по *IP* адресу.

9. Написать *GUI* приложение, которое представляет собой книгу заметок, позволяющую создавать новые заметки, редактировать и удалять существующие. Приложение позволяет создавать группы заметок и выполнять сортировку по дате.

10. Написать *GUI* приложение для проверки регулярных выражений. В двух полях пользователю требуется ввести текст и регулярное выражение. В поле текста после применения регулярного выражения требуется выделить найденную часть, если регулярное выражение написано с ошибкой, необходимо уведомить об этом пользователя.

11. Написать *GUI* приложение для работы с базой данных. Приложение позволяет выполнять соединение с БД, а также выполнять запросы и получать результаты ответов.

12. Написать *GUI* приложение для игры «Крестики-нолики».

13. Написать *GUI* приложение для игры «Сапер».

Лабораторная работа №10
**ВИЗУАЛИЗАЦИЯ РЕЗУЛЬТАТОВ РАБОТЫ
МАТЕМАТИЧЕСКИХ АЛГОРИТМОВ
С ИСПОЛЬЗОВАНИЕМ *NUMPY* И *MATPLOTLIB***

Цель работы: рассмотреть возможности пакета *NumPy*, его основные составляющие и возможности для обработки многомерных массивов данных; изучить библиотеку *Matplotlib* и её основные инструменты для создания графиков и диаграмм.

Краткая теория

NumPy – это пакет для научных расчётов на *Python*. Он содержит ряд особенностей:

- удобная и эффективная работа с обработкой многомерных данных;
- инструменты для интеграции кода на *C/C++*;
- работа с линейной алгеброй;
- поддержка огромного количества полезных алгоритмов.

Для установки данного пакета достаточно выполнить команду *pip install numpy* в командной строке или терминале, а для импорта его в скрипт соответственно *import numpy*.

Основным объектом рассматриваемого модуля является однородный многомерный массив (*ndarray*). Массивы аналогичны спискам, за исключением того, что они хранят только определенный тип данных, а работают они в несколько раз быстрее, чем списки.

Создать простой массив в *numpy* можно с помощью функции *numpy.array(x)*, где в качестве аргумента для параметра *x* можно передать список или кортеж. Стоит отметить, что данная функция возвращает объект типа *ndarray*. Вложенные списки или кортежи *numpy* сама формирует в многомерные массивы. На рис. 10.1 представлен пример создания многомерного массива в *numpy*.

Ниже приведены наиболее важные атрибуты объекта *ndarray*:

- *ndim*. Число измерений массива;
- *shape*. Размерность массива. Представляет собой кортеж, состоящий из двух чисел (*m,n*);

- *size*. Общее количество элементов массива;
- *dtype*. Тип элементов массива.

```
>>> import numpy as np
>>> array = np.array([[1,2,3,4], [1,2,3,4]])
>>> array
array([[1, 2, 3, 4],
       [1, 2, 3, 4]])
>>> type(array)
<class 'numpy.ndarray'>
>>>
```

Рис. 10.1. Пример создания многомерного массива с помощью *NumPy*

NumPy также предоставляет альтернативные способы создания массивов с помощью следующих функций:

- *zeros()* и *ones()* принимают кортеж с размерностью создаваемого массива и создают массив из нулей и из единиц соответственно;
- *arange()* аналогична функции *range*, но возвращает массив;
- *eye()* принимает число, задающее размерность и создает единичную матрицу. Данная функция предоставляет параметр *k*, в котором можно указать номер диагонали, где будут расположены единицы.

Кроме того, все перечисленные функции имеют параметр *dtype*, с помощью которого задается тип создаваемого массива.

Для работы с массивами можно использовать стандартные математические операции умножения, сложения, вычитания и т.д., но при этом стоит учитывать размерность массивов.

NumPy предоставляет функции для работы над массивами:

- *numpy.sum(arr)*. Принимает массив и возвращает сумму элементов массива;
- *numpy.prod(arr)*. Принимает массив и возвращает произведение элементов массива;
- *numpy.median(arr)*. Принимает массив и возвращает медиану;
- *ndarray.mean()*. Вызывается у объекта *ndarray* и возвращает среднее арифметическое значение элементов массива;

- *ndarray.min()*. Вызывается у объекта *ndarray* и возвращает минимальное значение в массиве;
- *ndarray.max()*. Вызывается у объекта *ndarray* и возвращает максимальное значение в массиве;
- *ndarray.argmin()*. Вызывается у объекта *ndarray* и возвращает индекс минимального элемента в массиве;
- *ndarray.argmax()*. Вызывается у объекта *ndarray* и возвращает индекс максимального элемента в массиве.

В арсенале *NumPy* присутствует ещё огромное количество инструментов для работы с линейной алгеброй, полиномами, статистикой, сортировкой, поиском и другим. Ознакомиться с полными возможностями модуля можно по ссылке [16].

Matplotlib – это *open-source* библиотека для построения графиков. С помощью данной библиотеки можно просто и быстро генерировать огромное количество различных графиков, гистограмм, диаграмм и многое другое, укладываясь всего лишь в несколько строк кода. С помощью модуля *pyplot* данная библиотека предоставляет интерфейс похожий на интерфейс *Matlab*. Данный модуль предоставляет полный контроль над стилями линий, свойствами шрифта и осей и т.д. через объектно-ориентированный интерфейс или набор функций, которые знакомы пользователям *Matlab*. *Matplotlib* поддерживает работу с *NumPy*.

Для установки данной библиотеки требуется выполнить команду *pip install matplotlib*.

В данной лабораторной работе будет рассмотрен модуль *pyplot*, который имеет следующие функции:

- *scatter(x, y)* – строит точку на графике с возможностью изменения размера маркера и цвета;
- *plot(arr_x, arr_y)* – строит ломаную линию, принимает в качестве аргументов массив координат по оси *X* и по оси *Y* соответственно;
- *text(x, y, str)* – добавляет текст на график. Функция принимает координаты текста по двум осям и саму строку, которую надо отобразить;
- *bar(arr_x, arr_height, width=0.8)* – создает столбчатую диаграмму. Принимает массив координат по оси *X* и мас-

сив высот, так же можно указать ширину графика, которая по умолчанию равно 0.8;

—`pie(arr_x, labels=None, colors=None)` – создает круговую диаграмму. Принимает массив размеров каждой из частей диаграммы, также дополнительно можно указать массив меток и цветов для каждой части.

Стоит отметить, что вышеприведенные функции на самом деле имеют гораздо большее число параметров, а здесь приведены только основные. Более подробно с модулем *pyplot* можно ознакомиться по ссылке [17].

Для создания нового графика требуется сначала создать объект *Figure*. Фигура представляет собой холст (*Canvas*), на котором отрисовывается график. Для удобства отображения при создании графиков можно создать сетку, вызвав функцию *grid()* и передав ей в качестве аргумента *True*. Для отображения созданной фигуры достаточно вызвать функцию *show()*. На рис. 10.2 приведен пример построения нескольких видов графиков.

```
import numpy as np
import matplotlib.pyplot as plt

rand_xy = np.random.randint(10, size=(2,10))

figure1 = plt.figure()
plt.plot(rand_xy[0], rand_xy[1])
plt.grid(True)

figure2 = plt.figure()
plt.bar(rand_xy[0], rand_xy[1])
plt.grid(True)

figure3 = plt.figure()
plt.pie(rand_xy[0], labels=rand_xy[0])

plt.show()
```

Рис. 10.2. Пример построения нескольких видов графиков массива с помощью *Matplotlib* и *NumPy*

Как видно из рис. 10.2, создаются три фигуры, на которых строятся три различных графика: ломаной линии, столбчатая диаграмма и круговая диаграмма. Данные генерируются случайно с помощью *NumPy*. Функции *randint()* передается макси-

мальное число и размерность генерируемого массива. Таким образом, создается три окна с различными графиками по одним и тем же данным. На рис. 10.3, 10.4 и 10.5 приведены созданные графики.

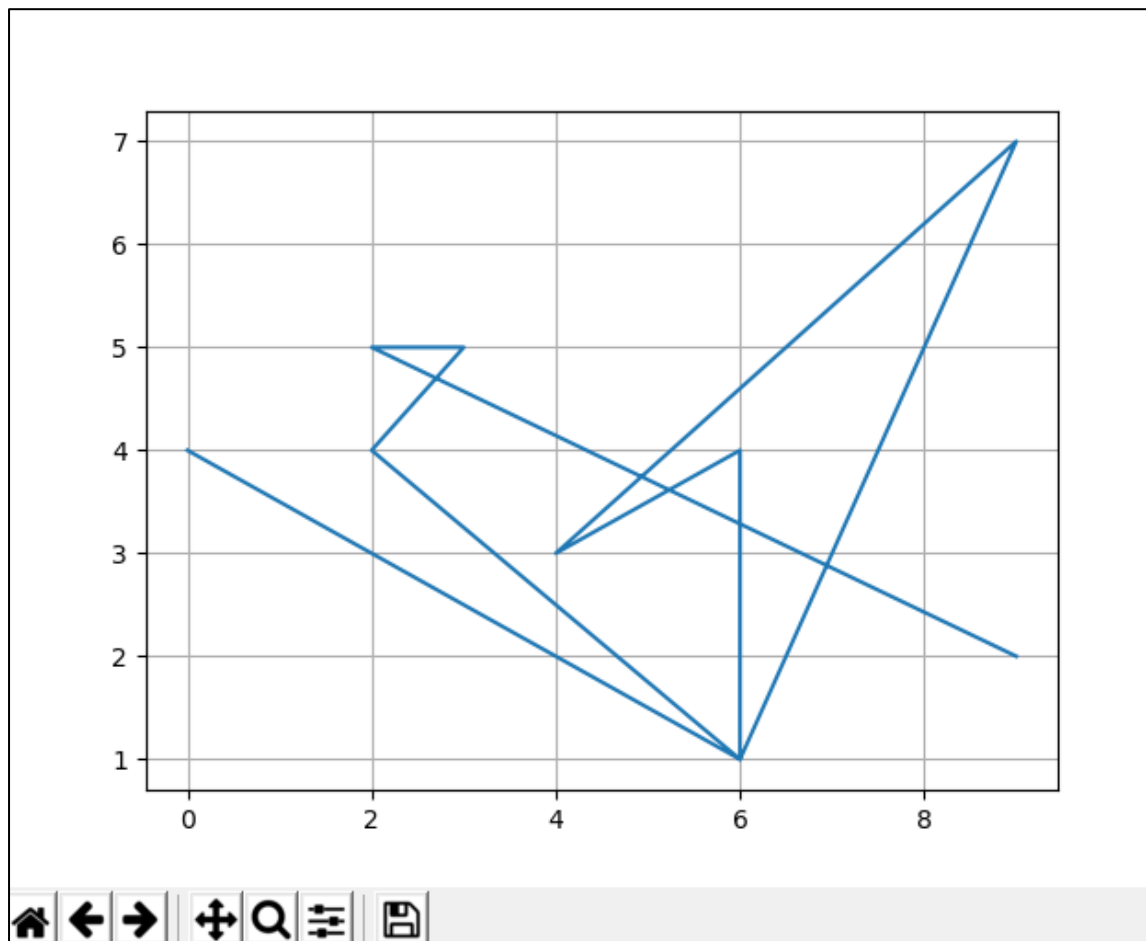


Рис. 10.3. Пример построения ломаной линии

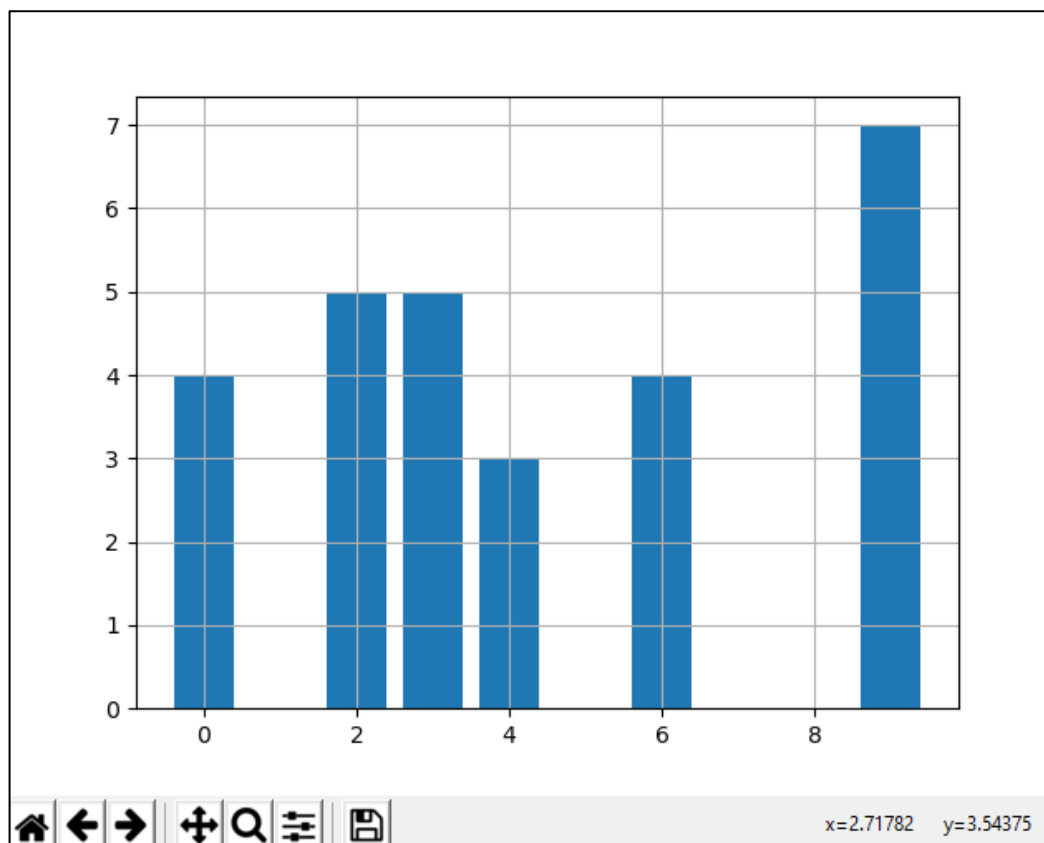


Рис. 10.4. Пример построения столбчатой диаграммы

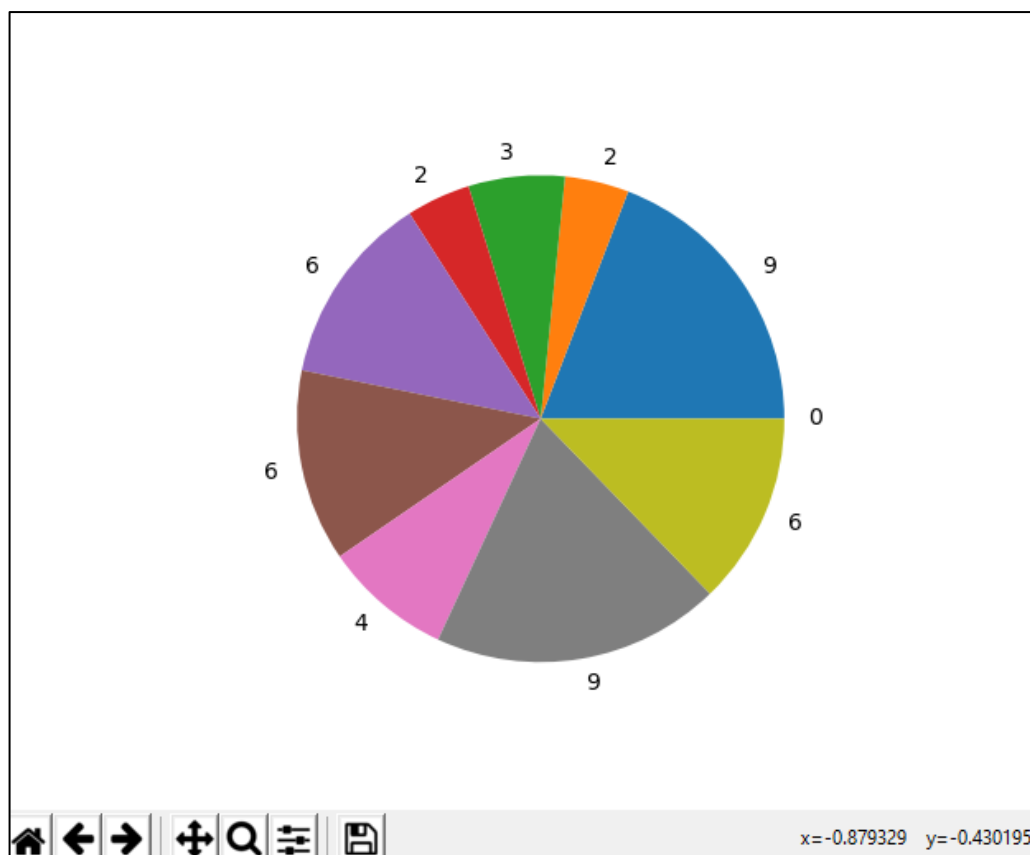


Рис. 10.5. Пример построения круговой диаграммы

Из приведенных рисунков построенных графиков видно, что на нижней панели окна расположены инструменты для изучения графиков, их настройка и возможность сохранения.

Таким образом, *Matplotlib* позволяет эффективно работать с отображением данных, применяя для этого минимум усилий. Данная библиотека, как и *NumPy* находит широкое применение не только в научных средах, но и для решения повседневных задач.

Варианты заданий

1. Написать функцию, которая принимает список целых чисел и возвращает *True*, если в списке отсутствуют нули, в противном случае возвращает *False*.

2. Написать функцию, которая создает массивы из десяти нулей, единиц и пятерок.

3. Написать функцию, которая создает массив из всех четных элементов между 50 и 90.

4. Написать функцию, которая принимает два вектора и возвращает их внешнее произведение.

5. Написать функцию, которая принимает вектор или матрицу и возвращает их норму.

6. Написать функцию, которая принимает матрицу и возвращает её определитель.

7. Написать функцию, которая принимает матрицу и возвращает сумму элементов её главной диагонали.

8. Написать функцию, которая генерирует шесть случайных чисел между 30 и 100.

9. Написать функцию, которая генерирует матрицу 5×5 и заполняет её случайными значениями и возвращает минимальное и максимальное значения полученной матрицы.

10. Написать функцию, которая принимает список целых чисел и возвращает список с перемешанными числами.

11. Написать функцию, которая строит гистограмму популярных языков программирования. Данные можно взять из последних данных *TIOBI*.

12. Написать функцию, которая читает таблицу из файла *.csv*, содержащую шапку и процентную долю каждого столбца и строит по ней круговую диаграмму.

13. Написать функцию, которая создает случайную точечную диаграмму со случайным размером шариков.

14. Написать функцию, которая строит точечную диаграмму повышения уровня моря за последние 100 лет.

Лабораторная работа №11

ОСНОВЫ РАБОТЫ С ВЕБ-ФРЕЙМВОРКОМ *DJANGO*

Цель работы: рассмотреть возможности веб-фреймворка *Django* для построения веб-приложений на языке *Python*, ознакомиться с понятием *ORM* и его реализации в данном фреймворке.

Краткая теория

Django – это веб-фреймворк для *Python*, основным преимуществом которого является высокая скорость разработки и «чистая» архитектура проекта. Данный фреймворк является *open-source* проектом. Архитектура *Django* предоставляет возможность быстрого и гибкого масштабирования. Данный фреймворк несет в себе множество дополнительного функционала «из коробки», такого как аутентификация пользователей, администрирования контента, панель администратора для работы с базой данных и многое другое.

Для работы с базой данных *Django*, как и многие другие веб-фреймворки используют *ORM* (*Object-Relational Mapping*). Это технология, позволяющая связать базы данных с парадигмой ООП. Данная технология позволяет не только не привязываться к определенной базе данных, но и генерировать архитектуру базы данных из написанных классов. В *ORM* предполагается, что каждая таблица является классом, а каждый атрибут (поле) класса является атрибутом таблицы. Таким образом, *ORM* представляет собой как бы прослойку между БД и ООП.

Создание проекта на *Django*

Перед началом создания проекта требуется установить сам веб-фреймворк. Это делают аналогично с установкой любой библиотеки через менеджера *pip*. Данная команда выполняет скачивание и установку фреймворка: *pip install django*.

После установки фреймворка можно приступить к созданию самого проекта. Для этого в командной строке или термине требуется перейти в нужную директорию, где будет создаваться проект и выполнить команду *django-admin*

startproject newproject, где *newproject* наименование создаваемого проекта. Данная команда создаст следующую структуру директорий и файлов:

- *newproject* – корневая директория проекта.
- *manage.py* – утилита командной строки, позволяющая взаимодействовать с проектом различными способами;
- *newproject* – директория, которая является основной для проекта и хранит в себе файл конфигурации (*settings.py*), менеджер *URL(urls.py)* и точку входа для *WSGI*-совместимых веб-серверов (*wsgi.py*).

В отличие от других платформ для разработки веб-приложений *Django* «из коробки» имеет собственный однопоточный веб-сервер для разработки приложения. Запустить на нем созданный проект можно, находясь в корневой директории проекта, с помощью команды *python manage.py runserver*. После чего при успешном старте сервера веб-приложение будет доступно по адресу <http://127.0.0.1:8000/>. Для остановки веб-сервера достаточно будет нажать комбинацию клавиш *Ctrl+C*, находясь в командной строке или терминале, где была введена предыдущая команда. Стоит отметить, что данный механизм не следует использовать на этапе работы приложения, а только во время его разработки.

Создание приложения в созданном проекте

Приложением называется модуль *Django* проекта. В них заключается мощь «чистой» архитектуры, которую предоставляет данный веб-фреймворк. Приложения удобно можно создавать самостоятельно, а можно заимствовать модули из иных проектов. Следует отметить, что для более правильной работы с предоставляемой архитектурой необходимо разделять создаваемые или заимствованные модули и стараться не допускать зависимостей между ними. Изолированные модули в будущем можно будет легко интегрировать в другие проекты, а ненужные удалять из проекта, не принося ущерба работоспособности приложения.

Итак, для того, чтобы создать новое приложение в созданном проекте необходимо, находясь в корневой директории, вы-

полнить команду `python manage.py startapp newapp`, где *newapp* наименование создаваемого приложения. Данная команда создаст новую директорию *newapp*, которая содержит следующую структуру:

- *migrations* – директория, которая в будущем будет содержать информацию о миграциях;
- *admin.py* – файл для настройки отображения созданных моделей на панели администратора;
- *apps.py* – файл для конфигурации самого приложения;
- *models.py* – файл для создания моделей;
- *tests.py* – файл для автоматизированных тестов;
- *views.py* – файл, содержащий бизнес-логику приложения.

К выше приведенной структуре следует и отнести файл *urls.py*, который создается вручную и аналогично одноименному файлу из основной директории проекта представляет собой менеджер *URL* для созданного приложения. После этого необходимо добавить созданное приложение в проект, добавив его наименование в список *INSTALLED_APPS* файла *settings.py*.

Работа с базой данных

Ранее было сказано о миграциях и моделях. Под моделями в *Django* следует понимать классы, описывающие таблицы базы данных. Стандартная модель должна наследоваться от класса *models.Model* и содержать определенный набор атрибутов, причем каждому атрибуту присваивается определенный тип. Стоит отметить, что атрибут суррогатного ключа (*id*) устанавливается по умолчанию. Ниже приведены несколько типов для объявления полей в моделях:

- *CharField()* – строковый тип, предназначенный для больших строк;
- *TextField()* – тип для хранения больших объемов текста;
- *IntegerField()*, *FloatField()* – типы для хранения целых чисел и числе с плавающей точкой соответственно;
- *ImageField()* – тип для хранения изображений;
- *EmailField()* – тип для хранения адресов электронной почты;

- *DateField()*, *DateTimeField()* – типы для хранения даты и даты и времени соответственно;
- *ForeignKey()* – тип для связки моделей по внешнему ключу. Требуется в качестве аргумента для параметра *to* наименование модели, с которой происходит связь (для рекурсивной зависимости следует использовать строковое значение *'self'*) и аргумент для параметра *on_delete* задающий логику действий при удалении зависимой записи.

Для примера в созданном проекте на рис. 11.1 показана модель для хранения статей и авторов.

Как видно из рис. 11.1, каждому типу полей передается определенный набор параметров. Подробно с каждым типом и возможными для него параметрами можно ознакомиться по ссылке [18]. После создания моделей и добавления созданного приложения в файл *settings.py* требуется выполнить миграцию для создания новых таблиц в базе данных. По умолчанию в *Django* задействована СУБД *SQLite*. Для этого в командной строке или терминале, находясь в корне проекта требуется последовательно выполнить две команды:

- *python manage.py makemigrations*;
- *python manage.py migrate*.

```
from django.db import models

# Create your models here.
class Author(models.Model):
    full_name = models.CharField(max_length=250)
    email = models.EmailField()

class Article(models.Model):
    author = models.ForeignKey(Author, on_delete=models.CASCADE)
    title = models.CharField(max_length=250, default='Unknown title')
    description = models.CharField(max_length=250)
    content = models.TextField()
    main_image = models.ImageField(upload_to='articlesimages')
    date_created = models.DateField()
```

Рис. 11.1. Пример создание моделей

Панель администратора

Для удобного взаимодействия с базой данных в *Django* предусмотрена панель администратора, которая устанавливается вместе с фреймворком. Для того, чтобы панель администратора была подключена к проекту, требуется проверить, что в файле *setting.py* указано приложение *django.contrib.admin*, а в файле *urls.py*, который находится в директории проекта, прописан путь *path('admin/', admin.site.urls)* к панели администратора.

Прежде чем перейти к работе, требуется создать суперпользователя, которому будут открыты все права на работу с базой данных. Для этого в командной строке или терминале, находясь в корневой директории проекта, требуется выполнить команду *python manage.py createsuperuser* и заполнить необходимые поля (поле *email* является не обязательным). После этого в файле *admin.py* созданного приложения требуется подключить созданные модели к панели администратора. На рис. 11.2 приведен пример добавления моделей на панель администратора.

```
from django.contrib import admin
from .models import *

# Register your models here.
admin.site.register(Author)
admin.site.register(Article)
```

Рис. 11.2. Пример добавления моделей на панель администратора

После этого можно запускать веб-сервер и переходить по ссылке <http://127.0.0.1:8000/admin/>. С помощью панели администратора можно добавлять, удалять записи в таблицах, а также управлять записями пользователей и группами. По умолчанию в базе уже будут созданы две таблицы *User* и *Group*. На рис. 11.3 приведен пример добавления записи в таблицу с помощью панели администратора.

Из рис. 11.3 можно заметить, что в качестве внешней записи для таблицы *Article* добавляется запись под названием *Author object(1)*, это стандартное имя записи в таблице. Отображаемое наименование записей можно изменить, добавив в классы моделей метод *__str()* как показано на рис. 11.4.

Django administration

WELCOME, ADMIN. [VIEW SITE](#) / [CHANGE PASSWORD](#) / [LOG OUT](#)

Home › Blog › Articles › Add article

Add article

Author: Author object (1) ▼ ✎ +

Title:

Description:

Content:

Содержание статьи

Main image: Choose File No file chosen

Date created: Today

Note: You are 3 hours ahead of server time.

Save and add another
Save and continue editing
SAVE

Рис. 11.3. Пример добавления записи в таблицу с помощью панели администратора

Как показано на рис. 11.4 после добавления метода `__str__()` для модели *Author* на панели администратора данные записи будут отображаться значением атрибута *full_name*.

```
from django.db import models

# Create your models here.
class Author(models.Model):
    full_name = models.CharField(max_length=250)
    email = models.EmailField()

    def __str__(self):
        return self.full_name

class Article(models.Model):
    author = models.ForeignKey(Author, on_delete=models.CASCADE)
    title = models.CharField(max_length=250, default='Unknown title')
    description = models.CharField(max_length=250)
    content = models.TextField()
    main_image = models.ImageField(upload_to='articlesimages')
    date_created = models.DateField()

    def __str__(self):
        return self.title
```

Рис. 11.4. Пример изменения стандартного наименования записи модели

Шаблоны

Под шаблонами в *Django* понимаются динамически сгенерированные *HTML* страницы. Шаблоны содержат как статическую часть, так и специальный синтаксис, описывающий, где и как будет располагаться динамический контент. Для этого используются шаблонизаторы. В *Django* можно использовать один или несколько шаблонизаторов. Встроенный шаблонизатор в *Django* называется *Django template language (DTL)*. Настройка поддержки *DTL* устанавливается при создании проекта.

Хранить создаваемые шаблоны необходимо в директориях под названием *templates*. Желательно хранить шаблоны, используемые приложениями в их директориях, а в самой директории *templates* создавать директорию с наименованием приложения. Это способствует устранению ошибок, т.к. при рендере шаблона фреймворк ищет его по указанному пути во всех директориях под названием *templates*.

Синтаксис *DTL* содержит четыре конструкции:

- *переменные*. Вывод значений из контекста, обрамляемый двумя фигурными скобками «{ { } }»;
- *тэги*. Предназначены для обеспечения логики в процессе рендера. Например, использование условий или циклов. Обрамляются фигурной скобкой и знаком процента «{ % % }»;
- *фильтры*. Трансформируют значения переменных и аргументы тэгов. Указываются через вертикальную черту после значения, которое необходимо изменить «|»
- *комментарии*. Однострочные обрамляются фигурной скобкой и знаком решетки «{ # # }», а многострочные выглядят следующим образом «{% comment %} ... {% comment %}»

В качестве примера на рис. 11.5 приведен вывод статей и их авторов на страницу.


```

<body>
  <div class="container">
    <h1 class="mb-3">Список статей</h1>
    {% for article in articles %}
      <div class="d-flex flex-row mb-3">
        <div class="card" style="width: 18rem;">
          <div class="card-body">
            <h5 class="card-title">{{ article.title }}</h5>
            <p class="card-text">{{ article.description }}</p>
            <p class="card-text">
              <small class="text-muted">
                {{ article.date_created }} | {{ article.author.full_name }}
              </small>
            </p>
            <a href="#" class="btn btn-primary">Прочитать</a>
          </div>
        </div>
      </div>
    {% endfor %}
  </div>
</body>

```

Рис. 11.5. Пример вывода статей и авторов на страницу с применением *DTL*

Представления

Однако для того, чтобы провести рендер страницы, приведенной на рис. 11.5, необходимо вызывать его из функции, находящейся в файле *views.py*. Функции, которые исполняются при вызове определенного *URL*, называются представлениями и указываются в файле *views.py*. Для этого, в файле *urls.py*, лежащем в корневой директории проекта, требуется указать адрес доступа к *urls*, находящимся в директории разработанного приложения. Например, в рассматриваемом примере требуется добавить в список *urlpatterns* строку *path('blog/', include('blog.urls'))*. После этого требуется указать адрес и функцию, которая будет выполняться при обращении к заданному адресу, как показано на рис. 11.6.

```

from django.urls import path

from . import views

urlpatterns = [
    path('', views.show_articles, name='articles'),
    path('article/<int:article_id>', views.show_article, name='article'),
]

```

Рис. 11.6. Пример добавления нового *URL* в приложение

Как видно из рис. 11.6 вторым аргументом для функции *path* является указание функции-представления в файле

views.py. В рассмотренном примере приведены два пути, первый вызывается при пути к приложению, а второй при добавлении к адресу *articles/* и идентификатор необходимой статьи.

На рис. 11.7 приведены реализации функций-представлений.

```
from django.shortcuts import render, get_object_or_404
from .models import *

# Create your views here.
def show_articles(request):
    articles = Article.objects.all()
    context = {'articles': articles}

    return render(request, 'blog/list-articles.html', context)

def show_article(request, article_id):
    article = get_object_or_404(Article, pk=article_id)

    return render(request, 'blog/article.html', {'article': article})
```

Рис. 11.7. Пример реализации функций-представлений

Из рис. 11.7 видно, что каждая создаваемая функция-представления, должна принимать объект *request*, который хранит информацию о запросе. В функции *show_articles()* происходит обращение к созданной модели *Article* и запрос на получение всех записей, аналогичных запросу *SELECT * FROM Articles* в *SQL*. Функция *show_article()* принимает в качестве аргумента для второго параметра значение идентификатора запрашиваемой статьи и с помощью вызова функции *get_object_or_404()* возвращает либо запись, либо перенаправляет на страницу с ошибкой 404.

На рис. 11.8 представлен пример вывода статьи на страницу с применением *DTL*.

```
<div class="container mt-5">
  <article>
    <h1 class="text-center">{{ article.title }}</h1>
    <p class="text-center"><b>{{ article.author.full_name }}</b></p>

    <div>
      <p>
        {{ article.content|safe }}
      </p>
    </div>

    <p>Дата публикации: <span>{{ article.date_created }}</span></p>
  </article>
</div>
```

Рис. 11.8. Пример вывода статьи на страницу с применением *DTL*

Как видно из рис. 11.8 для вывода содержания статьи указывается фильтр *safe*. Он позволяет экранировать текст, в котором содержаться *HTML* теги. Если данный фильтр отсутствует, то *HTML* выведется обычным текстом.

Результат обращения по адресу <http://127.0.0.1:8000/blog/> представлен на рис. 11.9. Страница с выводом статьи представлена на рис. 11.10.

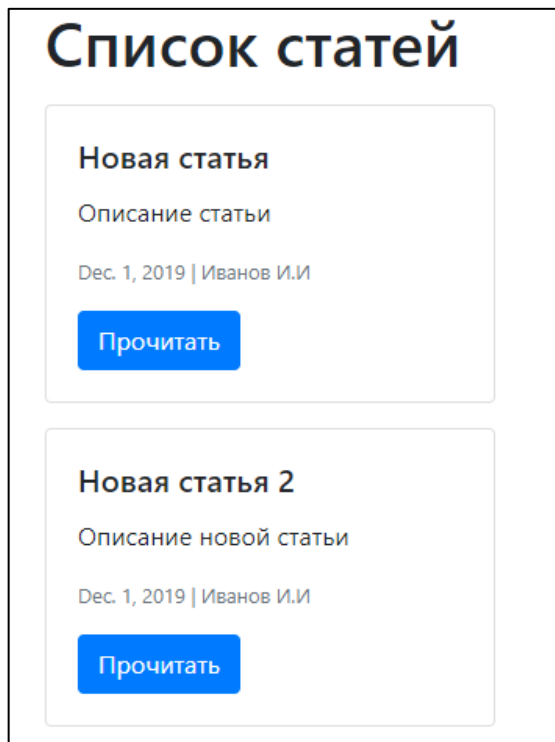


Рис. 11.9. Страница с выводом всех статей

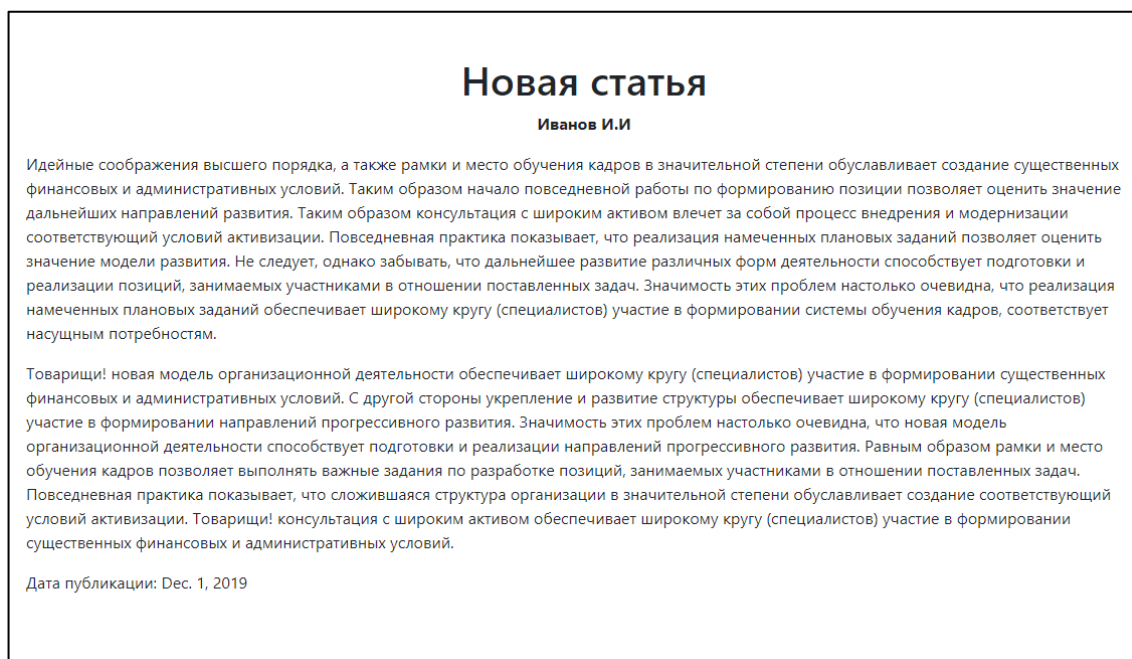


Рис. 11.10. Страница с выводом выбранной статьи

В работе с шаблонами хорошим приемом разработки *HTML* страниц в *Django* является использование тегов *extend* и *block* для многократного использования страниц без написания повторяющегося кода. Это особенно удобно, когда ко всем или многим страницам сайта требуется указать одни и те же *CSS* стили и *JavaScript* файлы.

В заключении стоит отметить, что стабильность *Django* и сообщество вокруг него выросли до невообразимых размеров с момента первого релиза. Официальная документация и учебные пособия по фреймворку являются одними из лучших в своём роде. А с каждой новой версией *Django* продолжает обрастать возможностями.

Варианты заданий

1. Написать веб-приложение, которое включает регистрацию пользователей на сайте, который выполняет случайную ежедневную рассылку вдохновляющих цитат на *e-mail* зарегистрированных пользователей.
2. Написать веб-приложение с небольшими вопросами (*flashcards*) наподобие *Quizlet*.
3. Написать веб-приложение небольшого интернет-блога с возможностью добавления новых статей, групп статей, редактирования и удаления существующих. Также приложение должно позволять выполнять поиск по наименованию статей, дате публикации и автору.
4. Написать веб-приложение простого интернет-магазина с каталогом товаров и добавлением товара в корзину.
5. Написать веб-приложение погоды с возможностью указания страны и города.
6. Написать веб-приложение для фитнеса, которое каждый день выдает очередную подборку тренировок для зарегистрированных пользователей. Типы тренировок можно выбирать пользователям в личном кабинете.

7. Написать веб-приложение книжного магазина, где для покупки требуется зарегистрироваться, а в личном кабинете можно будет прочитать купленные книги.

8. Написать веб-приложение музыкального сервиса, с возможностью прослушивания демо-версий музыкальных произведений и их покупки. Купленные песни в полной версии можно прослушать в личном кабинете. Также требуется реализовать поиск песни по исполнителю, году и наименованию.

9. Написать веб-приложение с использованием *django-rest-framework*. На стороне клиента для вывода данных требуется использовать один из *JavaScript* фреймворков (*Vue*, *Angular* или *React*).

10. Написать веб-приложение для непрерывного мониторинга какого-либо процесса. При этом должны выводиться графики, которые обновляются без перезагрузки страницы. Для решения этой задачи можно использовать *Ajax* или *JavaScript* фреймворки.

ЗАКЛЮЧЕНИЕ

Итак, *Python* – мощный высокоуровневый скриптовый язык программирования, используемый в разработке крупнейших платформ, сайтов и приложений. *Python* используется в крупных проектах из-за высокого качества программного обеспечения, кроссплатформенности, эффективности разработки, универсальности. Программисты ценят его за легкость в усвоении, простой и понятный синтаксис, удобочитаемость, большое количество библиотек, открытое сообщество. Популярность языка *Python* растет. Востребованность на рынке вакансий с каждым годом увеличивается.

В рамках лабораторных занятий по дисциплине «Программирование на языке *Python*» получено представление о базовом синтаксисе языка программирования *Python*, его основных возможностях, а также широкой функциональности, получаемой «из коробки».

Продемонстрирована гибкость и легкость в решении сложных задач с помощью данного языка. Приведены примеры использования различных внешних библиотек для решения различных задач.

В качестве среды разработки рассмотрена стандартная и гибкая, не слишком удобная для полноценной разработки, но достаточная для начальных разработок среда *IDLE*, поставляющаяся вместе с интерпретатором языка.

Были рассмотрены возможности легкого и удобного применения данного языка в различных задачах, таких как:

- работа с базами данных;
- обработка изображений;
- разработка приложений с графическим интерфейсом;
- применения языка в области математики и работы с данными;
- разработка веб-приложений.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Петров Ю. Python. Теория // URL: https://www.yuripetrov.ru/edu/python/ch_02_01.html (дата обращения: 21/12/2019).
2. The Incredible Growth of Python // StackOverflow URL: https://stackoverflow.blog/2017/09/06/incredible-growth-python/?utm_source=social&utm_medium=blog&utm_campaign=gen-blog&utm_content=blog-link&utm_term=why-python-growing-quickly (дата обращения: 21/12/2019).
3. Python is becoming the world's most popular coding language // The Economist. URL: <https://www.economist.com/graphic-detail/2018/07/26/python-is-becoming-the-worlds-most-popular-coding-language> (дата обращения: 21/12/2019).
4. Why is Python Growing So Quickly? // StackOverflow URL: <https://stackoverflow.blog/2017/09/14/python-growing-quickly/> (дата обращения: 21/12/2019).
5. How to Write Beautiful Python Code With PEP 8 // RealPython URL: <https://realpython.com/python-pep8/> (дата обращения: 21/12/2019).
6. Пол Бэрри. Изучаем программирование на Python. – М.: ООО «Издательство «Э»», 2017. – 624с.
7. Рейтц К., Шлюссер Т. Автостопом по Python. – СПб.: Питер, 2017. – 336с.
8. IDLE Documentation // python.org URL: <https://docs.python.org/3/library/idle.html> (дата обращения: 21/12/2019).
9. Data Structures Documentation // python.org URL: <https://docs.python.org/3/tutorial/datastructures.html> (дата обращения: 21/12/2019).
10. Built-in Functions Documentation // python.org URL: <https://docs.python.org/3/library/functions.html> (дата обращения: 21/12/2019).
11. Comprehensions // Python Tips URL: <http://book.pythontips.com/en/latest/comprehensions.html> (дата обращения: 21/12/2019).
12. Сулыз А.В. Синтаксический анализ веб-ресурсов // Новая наука: от идеи к результату. – 2016. – №10-3. – С. 108-109.
13. Python Programming/Reflection // Wikibooks URL: https://en.wikibooks.org/wiki/Python_Programming/Reflection (дата обращения: 21/12/2019).
14. Pillow Documentation // Pillow (PIL Fork) URL: <https://pillow.readthedocs.io/en/stable/> (дата обращения: 21/12/2019).
15. Graphical User Interfaces with Tk // python.org URL: Graphical User Interfaces with Tk (дата обращения: 21/12/2019).
16. NumPy Documentation// NumPy URL: <https://numpy.org/doc/> (дата обращения: 21/12/2019).
17. Matplotlib overview // Matplotlib URL: <https://matplotlib.org/3.1.1/contents.html> (дата обращения: 21/12/2019).
18. Django documentation// Django URL: <https://docs.djangoproject.com/en/3.0/> (дата обращения: 21/12/2019).

Учебное издание

**Широбокова Светлана Николаевна
Кацупеев Андрей Александрович
Сулыз Андрей Викторович**

**ПРОГРАММИРОВАНИЕ
НА ЯЗЫКЕ *PYTHON***

Учебное пособие
для лабораторных занятий

Редактор *Н.А. Юшко*

Подписано в печать 17.01.2020.

Формат 60×84 $\frac{1}{16}$. Бумага офсетная. Печать цифровая.

Усл. печ. л. 6,08. Уч.-изд. л. 6,25. Тираж 300 экз. Заказ 46-0258.

Южно-Российский государственный политехнический университет
(НПИ) имени М.И. Платова

Редакционно-издательский отдел ЮРГПУ(НПИ)
346428, г. Новочеркасск, ул. Просвещения, 132

Отпечатано в ИД «Политехник»
346428, г. Новочеркасск, пр. Первомайская, 166
idp-npi@mail.ru